Mentor Verification IP Altera Edition User Guide

Software Version 10.1_p3

April 2012

© 2012 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/
Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other third parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the respective third-party owner. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

Table of Contents

Chapter 1
Mentor VIP Altera Edition 13

Advantages of Using BFMs and Monitors 13
Implementation of BFMs 13
SystemVerilog API Overview 14
Configuration 15
set_config() 15
get_config 16
Creating Transactions 16
create*_transaction() 16
Executing Transactions 17
execute_transaction(), execute*_burst(), execute*_phase() 17
Waiting Events 17
wait_on() 17
Access Transaction 17
get*() 18
Operational Transaction Fields 18
Automatic Generation of Byte Lane Strobes 18
Operation Mode 18
Handshake Delay 19
Handshake Signal Delay Transaction Fields 19
Delay Mode 19
Data Beat Done 20
Transaction Done 21
AXI3 Protocol Restrictions 21
Simulator GCC Requirements 21

Chapter 2
SystemVerilog AMBA AXI3 BFMs 23

AMBA AXI3 Master BFM 23
AXI3 Master Timing 23
AXI3 Master Configuration 24
AXI3 Master Assertions 25
SystemVerilog AXI3 Master Application Program Interface 25
set_config() 26
get_config() 27
create_write_transaction() 28
create_read_transaction() 31
execute_transaction() 34
execute_write_addr_phase() 35
execute_read_addr_phase() 36
execute_write_data_burst() 37

execute_write_data_phase() . . . 38
get_read_data_burst() . . . 39
get_read_data_phase() . . . 40
get_write_response_phase . . . 41
wait_on() . . . 42

Chapter 3
SystemVerilog AMBA AXI3 Slave BFM . . . 43

AXI3 Slave Timing . . . 43
AXI3 Slave Configuration . . . 44
AXI3 Slave Assertions . . . 45
SystemVerilog AXI3 Slave Application Program Interface . . . 45
set_config() . . . 46
get_config() . . . 47
create_slave_transaction() . . . 48
execute_read_data_burst() . . . 49
execute_read_data_phase() . . . 50
execute_write_response_phase() . . . 51
get_write_addr_phase() . . . 52
get_read_addr_phase() . . . 53
get_write_data_phase() . . . 54
get_write_data_burst() . . . 55
wait_on() . . . 56
Helper Functions . . . 56
get_write_addr_data() . . . 57
get_read_addr() . . . 58
set_read_data() . . . 59

Chapter 4
SystemVerilog AMBA AXI3 Monitor BFM . . . 61

AXI3 Inline Monitor Connection . . . 61
AXI3 Monitor Timing . . . 61
AXI3 Monitor Configuration . . . 62
AXI3 Monitor Assertions . . . 64
SystemVerilog AXI3 Monitor Application Program Interface . . . 64
set_config() . . . 65
get_config() . . . 66
create_monitor_transaction() . . . 67
get_rw_transaction() . . . 68
get_write_addr_phase() . . . 71
get_read_addr_phase() . . . 73
get_write_data_phase() . . . 75
get_write_data_burst() . . . 76
get_write_response_phase . . . 77
get_read_data_phase() . . . 78
get_read_data_burst() . . . 79
wait_on() . . . 80
Helper Functions . . . 80

get_write_addr_data() 81
get_read_addr() 82
set_read_data() 83

Chapter 5
AXI3 Assertions 85

Chapter 6
SystemVerilog Tutorials 97

Verifying a Slave DUT 97
Master Test Program 98
Verifying a Master DUT 100
Slave Test Program 101

Chapter 7
VHDL API Overview 111

Configuration 113
set_config() 113
get_config 113
Creating Transactions 113
create*_transaction() 114
Executing Transactions 114
execute_transaction(), execute*_burst(), execute*_phase() 114
Waiting Events 115
wait_on() 115
Access Transaction 115
get*() 115
Operational Transaction Fields 115
Automatic Generation of Byte Lane Strobes 115
Operation Mode 116
Handshake Delay 116
Handshake Signal Delay Transaction Fields 116
Delay Mode 117
Data Beat Done 118
Transaction Done 118

Chapter 8
AMBA AXI VHDL BFMs 119

Overloaded Procedure Common Arguments 119
AMBA AXI3 Master BFM 119
AXI3 Master Timing 119
AXI3 Master Configuration 120
AXI3 Master Assertions 121
VHDL AXI3 Master Application Program Interface 121
set_config() 122
get_config() 123
create_write_transaction() 124
create_read_transaction() 127

execute_transaction() . . . 129
execute_write_addr_phase() . . . 130
execute_read_addr_phase() . . . 131
execute_write_data_burst() . . . 132
execute_write_data_phase() . . . 133
get_read_data_burst() . . . 134
get_read_data_phase() . . . 135
get_write_response_phase . . . 136
wait_on() . . . 137

Chapter 9
VHDL AMBA AXI3 Slave BFM . . . 139

VHDL AXI3 Slave Timing . . . 139
VHDL AXI3 Slave Configuration . . . 139
VHDL AXI3 Slave Assertions . . . 141
VHDL AXI3 Slave Application Program Interface . . . 141
set_config() . . . 142
get_config() . . . 143
create_slave_transaction() . . . 144
execute_read_data_burst() . . . 145
execute_read_data_phase() . . . 146
execute_write_response_phase() . . . 147
get_write_addr_phase() . . . 148
get_read_addr_phase() . . . 149
get_write_data_phase() . . . 150
get_write_data_burst() . . . 151
wait_on() . . . 152
Helper Functions . . . 152
get_write_addr_data() . . . 153
get_read_addr() . . . 154
set_read_data() . . . 155

Chapter 10
VHDL AMBA AXI3 Monitor BFM . . . 157

VHDL AXI3 Inline Monitor Connection . . . 157
VHDL AXI3 Monitor Timing . . . 157
VHDL AXI3 Monitor Configuration . . . 158
AXI3 Monitor Assertions . . . 159
VHDL AXI3 Monitor API . . . 159
set_config() . . . 160
get_config() . . . 161
create_monitor_transaction() . . . 162
get_rw_transaction() . . . 163
wait_on() . . . 166

Chapter 11
VHDL Tutorials . . . 167

Verifying a VHDL Slave DUT . . . 167

VHDL Master Test Program 168
Verifying a VHDL Master DUT 170
VHDL Slave Test Program 171
VHDL Basic Slave API Definition 172
VHDL Advanced Slave API Definition 177

Chapter 12
Getting Started with Qsys and the BFMs 183

Setting Up the Simulation Environment 183
Running Simulation 187
Invoking Simulation From a GUI 188
Invoking Simulation From a Command Line 189
Example Script Processing 190
Using a Shortcut or Editing the modelsim.ini File 191

Appendix A
SystemVerilog
Master and Slave Test Programs 193

SystemVerilog Master Test Program 193
SystemVerilog Slave Test Program 197

Appendix B
VHDL Master and Slave Test Programs 203

VHDL Master BFM Code Example 203
VHDL Slave BFM Code Example 208

List of Tables

Table 1-1. Handshake Signal DelayTransaction Fields 19
Table 1-2. Simulator GCC Requirements 22
Table 2-1. AXI3 Master BFM Configuration 24
Table 2-2. SystemVerilog AXI3 Master set_config() 27
Table 2-3. SystemVerilog AXI3 Master get_config() 28
Table 2-4. SystemVerilog AXI3 Master create_write_transaction Function 29
Table 2-5. SystemVerilog AXI3 Master create_read_transaction Function 32
Table 2-6. SystemVerilog AXI3 Master execute_transaction Task 35
Table 2-7. SystemVerilog AXI3 Master execute_write_addr_phase Task 36
Table 2-8. SystemVerilog AXI3 Master execute_read_addr_phase Task 37
Table 2-9. SystemVerilog AXI3 Master execute_write_data_burst Task 38
Table 2-10. SystemVerilog AXI3 Master execute_write_data_phase Task 39
Table 2-11. SystemVerilog AXI3 Master get_read_data_burst Task 40
Table 2-12. SystemVerilog AXI3 Master get_read_data_phase Task 41
Table 2-13. SystemVerilog AXI3 Master get_write_response_phase Task 42
Table 2-14. SystemVerilog AXI3 Master wait_on Task 43
Table 3-1. AXI3 Slave BFM Configuration 46
Table 3-2. SystemVerilog AXI3 Slave set_config() 49
Table 3-3. SystemVerilog AXI3 Slave get_config() 50
Table 3-4. SystemVerilog AXI3 Slave create_slave_transaction Function 51
Table 3-5. SystemVerilog AXI3 Slave execute_read_data_burst Task 52
Table 3-6. SystemVerilog AXI3 Slave execute_read_data_phase Task 53
Table 3-7. SystemVerilog AXI3 Slave execute_write_response_phase Task 54
Table 3-8. SystemVerilog AXI3 Slave execute_write_response_phase Task 55
Table 3-9. SystemVerilog AXI3 Slave get_write_addr_phase Task 56
Table 3-10. SystemVerilog AXI3 Slave get_write_data_phase Task 57
Table 3-11. SystemVerilog AXI3 Slave get_write_data_burst Task 58
Table 3-12. SystemVerilog AXI3 Slave wait_on Task 59
Table 3-13. SystemVerilog AXI3 Slave get_write_addr_data Function 60
Table 3-14. SystemVerilog AXI3 Slave get_write_addr Function 61
Table 3-15. SystemVerilog AXI3 Slave get_write_addr_data Function 62
Table 4-1. AXI3 Monitor BFM Configuration 64
Table 4-2. SystemVerilog AXI3 Monitor set_config() 67
Table 4-3. SystemVerilog AXI3 Monitor get_config() 68
Table 4-4. SystemVerilog AXI3 Monitor create_monitor_transaction Function 69
Table 4-5. SystemVerilog AXI3 Monitor get_rw_transaction Task 70
Table 4-6. SystemVerilog AXI3 Monitor get_write_addr_phase Task 73
Table 4-7. SystemVerilog AXI3 Monitor get_write_addr_phase Task 75
Table 4-8. SystemVerilog AXI3 Monitor get_write_data_phase Task 78
Table 4-9. SystemVerilog AXI3 Monitor get_write_data_burst Task 79

Table 4-10. SystemVerilog AXI3 Monitor get_write_response_phase Task 80
Table 4-11. SystemVerilog AXI3 Monitor get_read_data_phase Task 81
Table 4-12. SystemVerilog AXI3 Monitor get_read_data_burst Task 82
Table 4-13. SystemVerilog AXI3 Monitor wait_on () 83
Table 4-14. SystemVerilog AXI3 Monitor get_write_addr_data Function 84
Table 4-15. SystemVerilog AXI3 Monitor get_read_addr Function 85
Table 4-16. SystemVerilog AXI3 Monitor set_read_data Function 86
Table 5-1. Assertions 87
Table 7-1. Handshake Signal Delay Transaction Fields 120
Table 8-1. AXI3 Master BFM Configuration 124
Table 8-2. VHDL AXI3 Master set_config() Procedure 127
Table 8-3. VHDL AXI3 Master get_config() Procedure 128
Table 8-4. VHDL AXI3 Master create_write_transaction Procedure 129
Table 8-5. VHDL AXI3 Master create_read_transaction Procedure 132
Table 8-6. VHDL AXI3 Master execute_transaction Procedure 135
Table 8-7. VHDL AXI3 Master execute_write_addr_phase Procedure 136
Table 8-8. VHDL AXI3 Master execute_read_addr_phase Procedure 137
Table 8-9. VHDL AXI3 Master execute_write_data_burst Procedure 138
Table 8-10. VHDL AXI3 Master get_read_data_burst Task 140
Table 8-11. VHDL AXI3 Master get_read_data_phase Task 141
Table 8-12. VHDL AXI3 Master get_write_response_phase Procedure 142
Table 8-13. VHDL AXI3 Master wait_on Procedure 143
Table 9-1. VHDL AXI3 Slave BFM Configuration 145
Table 9-2. VHDL AXI3 Slave set_config() Procedure 148
Table 9-3. VHDL AXI3 Slave get_config() Procedure 149
Table 9-4. VHDL AXI3 Slave create_slave_transaction Function 150
Table 9-5. VHDL AXI3 Slave execute_read_data_burst Procedure 151
Table 9-6. VHDL AXI3 Slave execute_read_data_phase Procedure 152
Table 9-7. VHDL AXI3 Slave execute_write_response_phase Procedure 153
Table 9-8. VHDL AXI3 Slave get_write_addr_phase Procedure 154
Table 9-9. VHDL AXI3 Slave get_read_addr_phase Procedure 155
Table 9-10. VHDL AXI3 Slave get_write_data_phase Procedure 156
Table 9-11. VHDL AXI3 Slave get_write_data_burst Procedure 157
Table 9-12. VHDL AXI3 Slave wait_on Procedure 158
Table 9-13. VHDL AXI3 Slave get_write_addr_data Procedure 159
Table 9-14. VHDL AXI3Slave get_read_addr Procedure 160
Table 9-15. VHDL AXI3Slave get_write_addr_data Procedure 161
Table 10-1. AXI3 Monitor BFM Configuration 164
Table 10-2. VHDL AXI3 Monitor set_config() Procedure 167
Table 10-3. VHDL AXI3 Monitor get_config() Procedure 168
Table 10-4. VHDL AXI3 Monitor create_monitor_transaction Procedure 169
Table 10-5. VHDL AXI3 Monitor get_rw_transaction Procedure 170
Table 10-6. VHDL AXI3 Monitor wait_on Procedure 173
Table 12-1. Simulator and Script Names 195

List of Figures

Figure 1-1. SystemVerilog BFM Internal Structure 15
Figure 1-2. Operational Transaction Field delay_mode = AXI_VALID2READY. 20
Figure 1-3. Operational Transaction Field delay_mode = AXI_TRANS2READY 20
Figure 4-1. Inline Monitor Connection Diagram. 61
Figure 6-1. Slave DUT Top-level Testbench Environment. 97
Figure 6-2. Master DUT Top-level Testbench Environment. 100
Figure 6-3. API Pipelining. 105
Figure 7-1. VHDL BFM Internal Structure 112
Figure 7-2. Operational Transaction Field delay_mode = AXI_VALID2READY. 117
Figure 7-3. Operational Transaction Field delay_mode = AXI_TRANS2READY 118
Figure 10-1. Inline Monitor Connection Diagram. 157
Figure 11-1. Slave DUT Top-level Testbench Environment. 167
Figure 11-2. Master DUT Top-level Testbench Environment. 170
Figure 11-3. API Pipelining. 171
Figure 12-1. Select Qsys from the Quartus II Software Top-Level Menu 184
Figure 12-2. Select and Open the Example File 184
Figure 12-3. Quartus II Software Displays Connectivity of the Example 185
Figure 12-4. Generation Tab Window 185
Figure 12-5. Set Path, Simulation and Synthesis Options 186
Figure 12-6. Click Generate 187
Figure 12-7. Select the Work Directory. 188

Preface

About This User Guide

This Mentor® Verification IP (VIP) Altera ® Edition (AE) User Guide describes the application interface (API) of the Mentor VIP AE and how it conforms to the AMBA® AXI Protocol Specification, Version 2.0.

AMBA AXI Protocol Specification

The Mentor VIP AE conforms to the AMBA® AXI™ Protocol Specification, Version 2.0 (available from ARM). Restrictions to this protocol are described in the section "AXI3 Protocol Restrictions" on page 21.

This user guide refers to the AMBA® AXI™ Protocol Specification, Version 2.0. as the AMBA AXI Protocol Specification.

Supported Simulators

Mentor VIP AE supports the following simulators:

- Mentor Graphics Modelsim (including Altera Editions) and Questa Sim 10.0d
- Synopsys VCS and VCS-MX 2011.12
- Cadence IUS 11.1

Chapter 1 Mentor VIP Altera Edition

The Mentor® Verification IP (VIP) Altera® Edition (AE) provides bus functional models (BFMs) to simulate the behavior and to facilitate the verification of IP. The Mentor VIP AE includes the following interfaces:

AXI3™ BFM with master, slave, and inline monitor interfaces

Advantages of Using BFMs and Monitors

Using the Mentor VIP AE has the following advantages:

- It accelerates the verification process by providing key components of the verification testbench.
- It provides BFM components that implement the AMBA AXI Protocol Specification, which serves as a reference for the protocol.
- It provides a full suite of configurable assertion checking within each BFM.

Implementation of BFMs

The Mentor VIP AE BFMs, master, slave, and inline monitor components are implemented in SystemVerilog. Also included are wrapper components so that the BFMs can be used in VHDL verification environments with simulators that support mixed-language simulation.

The Mentor VIP AE provides you with a set of APIs for each BFM that you can use to construct, instantiate, control, and query signals in all BFM components. Your test programs must use only these public access methods and events to communicate with each BFM. Your test programs must use the standard set of APIs to interface with the BFMs to ensure support in current and future releases. Nonstandard APIs and user-generated interfaces may not be supported in future releases.

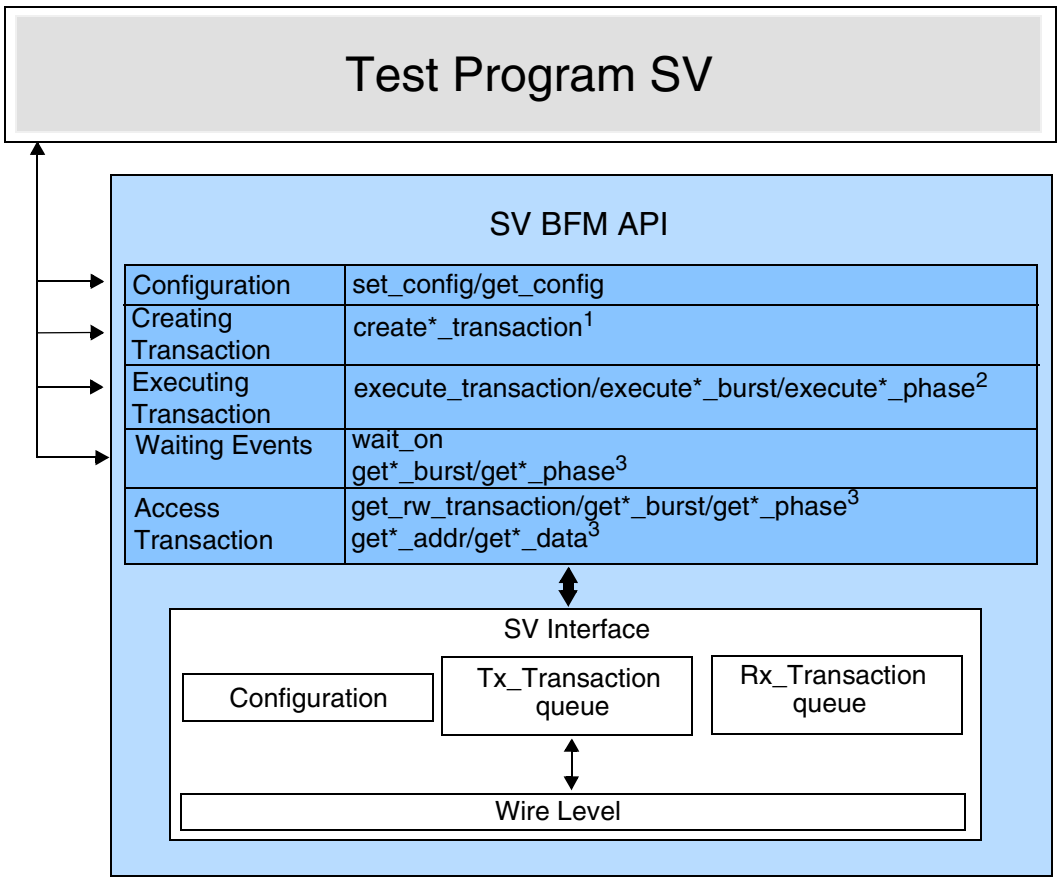
The test program drives the stimulus to the DUTs and determines whether the DUTs' behavior is correct by analyzing the responses. The BFMs translate the test program stimuli, creating the signaling for the AMBA AXI Protocol Specification. The BFMs also check for protocol compliance by firing an assertion when a protocol error is observed.

SystemVerilog API Overview

This section provides the functional description of the SystemVerilog Application Programming Interface (API) for all the BFM (master, slave, and monitor) components. For each BFM, you can configure the protocol transaction fields that are executed on the protocol signals, as well as control the operational transaction fields that permit delays to be introduced between the handshake signals for each of the five address, data, and response channels.

In addition, the BFMs permit you to wait for certain events to occur on the system clock and reset signals, plus get/set information about a particular transaction.

Figure 1-1. SystemVerilog BFM Internal Structure



Notes: 1. Refer to create*_transaction()
2. Refer to execute_transaction(), execute*_burst(), execute*_phase()
3. Refer to get*()

Configuration

Configuration involves the setting of timing configuration, error reporting, and other attributes of the BFM.

set_config()

Each master, slave, and monitor BFM has a *set_config()* function that sets the configuration of the BFM. Refer to the individual BFM *set_config()* tables for valid configurations.

For example, to set the burst timeout factor for a transaction:

```
// Setting the burst timeoutfactor to1000
   master_bfm.set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config

Each master, slave, and, monitor BFM has a *get_config()* function that returns the configuration of BFM. Refer to the individual *get_config* tables for details.

For example, to get the protocol signal hold time:

```
// Getting hold time value
   hold_time = master_bfm.get_config(AXI_CONFIG_HOLD_TIME);
```

Creating Transactions

An AXI3 transaction consists of operations occurring on the bus wires, which are parameterized and defined by specifying the address, length, data, size, strobes, etc. The master API allows you to create a master transaction by providing only the address and burst length arguments for a read, or write, transaction. All other protocol transaction fields automatically default to legal AXI3 protocol values to create a complete transaction. If you change a protocol transaction field value from its default, it is then valid for all future transactions until a new value is set. Refer to *create_write_transaction()* for default protocol transaction field values.

The slave API also allows you to create a slave transaction by providing no arguments. All protocol transaction fields automatically default to legal AXI3 protocol values to create a complete slave transaction. If you change a protocol transaction field value from its default, it is then valid for all future transactions until a new value is set. Refer to the section "create_slave_transaction()" on page 51 for default protocol transaction field values.

create*_transaction()

There are two master API functions available to create AXI3 transactions, *create_write_transaction()* and *create_read_transaction()*, a single slave API function c*reate_slave_transaction()* and a single monitor API function *create_monitor_transaction()*.

For example, to create a simple write transaction with a start address of 1 with a single beat data burst with a data value of 2:

```
// Define a variable trans of type axi_transaction
    axi_transaction trans;
```

```
// Create master write transaction
    trans = bfm.create_write_transaction(1);
    trans.data_words[0]   = 2;
```

For example, to create a simple slave transaction:

```
// Define a variable write_trans of type axi_transaction
    axi_transaction write_trans;
// Create slave transaction
    write_trans = bfm.create_slave_transaction();
```

Executing Transactions

Tasks create transactions that when executed are pushed into internal transaction queues. Figure 1-1 illustrates the execution onto protocol signals.

execute_transaction(), execute*_burst(), execute*_phase()

Typically, you use the *execute_transaction()* task at the master end and use *execute*_burst()* and *execute*_phase()* tasks at the slave end.

For example, to execute a master write transaction:

```
// By default the execution of a transaction will block
    bfm.execute_transaction(trans);
```

For example, to execute a slave transaction:

```
// By default the execution of a transaction will block
    bfm.execute_write_response_phase(write_trans);
```

Waiting Events

Each master, slave and monitor BFM has a *wait_on()* task that permits the user to wait for an AXI3 *ACLK,* or *ARESETn*, event to occur before proceeding.

wait_on()

For example, for the master to wait on the positive edge of the *ARESETn* signal:

```
// Block test program execution until the positive edge of the clock
    bfm.wait_on(AXI_RESET_POSEDGE);
```

Access Transaction

Each master, slave and monitor BFM can access a complete, or partially complete, transaction by using various *get*()* tasks to gather information.

get*()

For example, a slave may use a received write address phase to form the response of the transaction.

```
write_trans = bfm.create_slave_transaction();
bfm.get_write_addr_phase(write_trans);

  …

bfm.execute_write_response_phase(write_trans);
```

Operational Transaction Fields

Operational transaction fields control the way in which a transaction is executed onto the protocol signals and not the content, with the exception of the write strobes automatic generation transaction field. They also provide an indicator of when a data phase (beat), or a complete transaction, has completed.

Automatic Generation of Byte Lane Strobes

The AXI3 protocol permits unaligned and narrow write transfers by using byte lane strobe (*WSTRB)* signals to indicate which byte lanes contain valid data per data phase (beat).

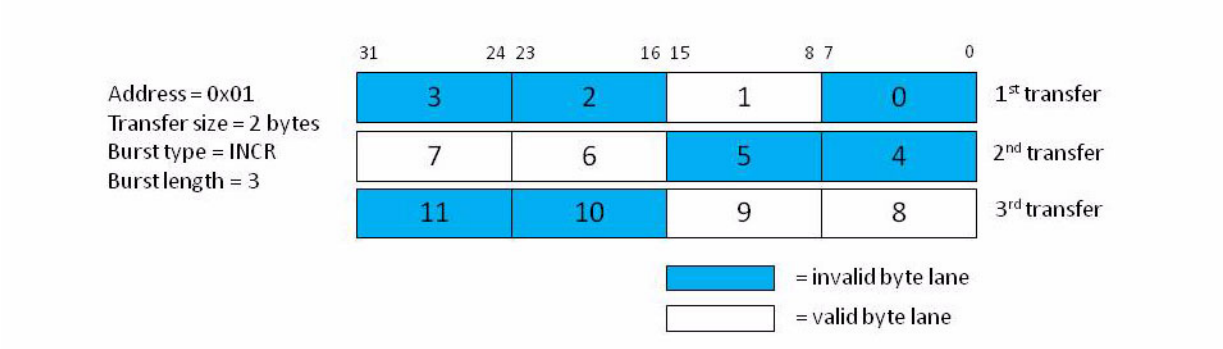
When you create a write transaction in your test program, the array variable *write_strobes[]* is available to store the write strobe values for each write data phase (beat) within the transaction. To assist you in creating the correct byte lane strobes per write data phase (beat), automatic correction of previously set *write_strobes[]* array elements is performed by default during execution of the write transaction or write data burst. You can disable this default behavior by setting the transaction field *gen_write_strobes = 0,* which allows any previously set *write_strobes[]* array elements to pass through uncorrected onto the protocol *WSTRB* signals. In this mode, with the automatic correction disabled, you are responsible for setting the correct *write_strobes[]* array elements for the whole transaction.

The automatic correction algorithm performs a bit-wise AND operation on any previously set *write_strobes[]* array elements. To do the corrections, the correction algorithm uses the equations described in the AMBA AXI Protocol Specification, version 2.0, section 4.5 that define valid write data byte lanes for legal protocol. Therefore, if you require automatic generation of all *write_strobes[]* array elements for a write transaction you must previously set all *write_strobes[]* array elements to be ‘1’, indicating that all bytes lanes initially contain valid

write data, prior to execution of the write transaction. Automatic correction will then set the relevant *write_strobes[]* array elements to '0' to produce legal protocol *WSTRB* signals.

For example, Figure 1-2 below shows byte lanes that may contain valid data for a write transaction with a starting address = 0x01, size = 0b001 (2 bytes), type = INCR, and the length = 0b0010 (3 beats) for a 32-bit, write data bus.

Figure 1-2. Valid Data on Byte Lanes During a Write Transaction



In the above example, if you set all *write_strobes[]* array elements to 1 prior to executing the write transaction, automatic correction produces the following result while the transaction executes.

	Prior to execution		During execution
1st data phase -	write_strobes[0]=0b1111	->	write_strobes[0]=0b0010
2nd data phase -	write_strobes[1]=0b1111	->	write_strobes[1]=0b1100
3rd data phase -	write_strobes[2]=0b1111	->	write_strobes[2]=0b0011

If you randomly set all *write_strobes[]* array elements to 0 or 1, prior to exciting the write transaction, automatic correction **only** corrects those *write_strobes[]* array elements that were previously set to 1, as shown below.

	Prior to execution		During execution
1st data phase -	write_strobes[0]=0b1010	->	write_strobes[0]=0b0010
2nd data phase -	write_strobes[1]=0b1010	->	write_strobes[1]=0b1000
3rd data phase -	write_strobes[2]=0b1010	->	write_strobes[2]=0b0010

Note

To automatically generate all *WSTRB* signals for a write transaction, set all *write_strobes[]* array elements to 1 prior to execution of the write transaction or write data burst.

Operation Mode

By default, each read or write transaction will perform a blocking operation which prevents the following transaction from starting until the current active transaction has completed.

You can configure this behavior to be nonblocking by setting the *operation_mode* transaction field to the enumerate type value *AXI_TRANSACTION_NON_BLOCKING* instead of the default *AXI_TRANSACTION_BLOCKING*. Typically, this configuration is performed by calling the "create_write_transaction()" on page 29 and "create_read_transaction()" on page 32 tasks at the master BFM and the "create_slave_transaction()" on page 51 at the slave BFM.

Handshake Delay

You can configure the delay between the **VALID* and **READY* handshake signals for each of the five address, data, and response channels. These delays can be set on a per phase cycle basis for a particular transaction, measured from the positive edge of *ACLK* when **VALID* is asserted within a transaction, or the completion of a previous transaction phase (**VALID* and **READY* both asserted).

Handshake Signal Delay Transaction Fields

There are a number of transaction fields that you can configure to produce the desired handshake delay pattern for a particular transaction phase on any of the five protocol channels. The master BFM of the protocol configures the **VALID* and **READY* signal delays that a master asserts, with the slave BFM being responsible for the configuration of the **VALID* and **READY* signal delays that it asserts. Table 1-1 details which operational delay transaction field is configured by the master/slave BFM.

Table 1-1. Handshake Signal Delay Transaction Fields

Signal	Operational Transaction Field	Configuration BFM
AWVALID	address_valid_delay	Master
AWREADY	address_ready_delay	Slave
WVALID	data_valid_delay	Master
WREADY	data_ready_delay	Slave
BVALID	write_response_valid_delay	Slave
BREADY	write_response_ready_delay	Master
ARVALID	address_valid_delay	Master
ARREADY	address_ready_delay	Slave
RVALID	data_valid_delay	Slave
RREADY	data_ready_delay	Master

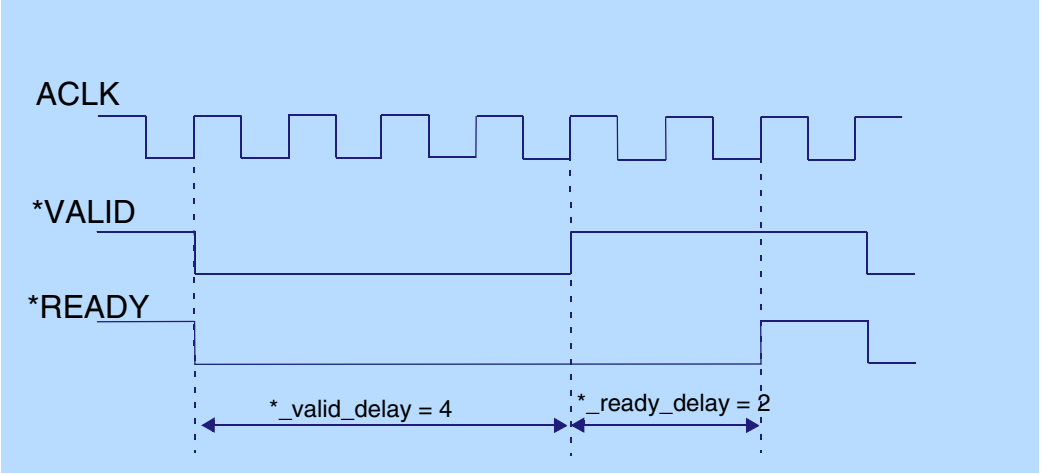
The data channel handshake signal transaction fields (*data_valid_delay[]* and *data_ready_delay[]*) are defined as arrays so that the **VALID* to **READY* delay can be configured on a per data phase (beat) basis within a transaction.

Delay Mode

The delay mode can be configured on a per transaction basis via the *delay_mode* operational transaction field. This transaction field can be configured to the enumerated type values of *AXI_VALID2READY* (default), or *AXI_TRANS2READY*.

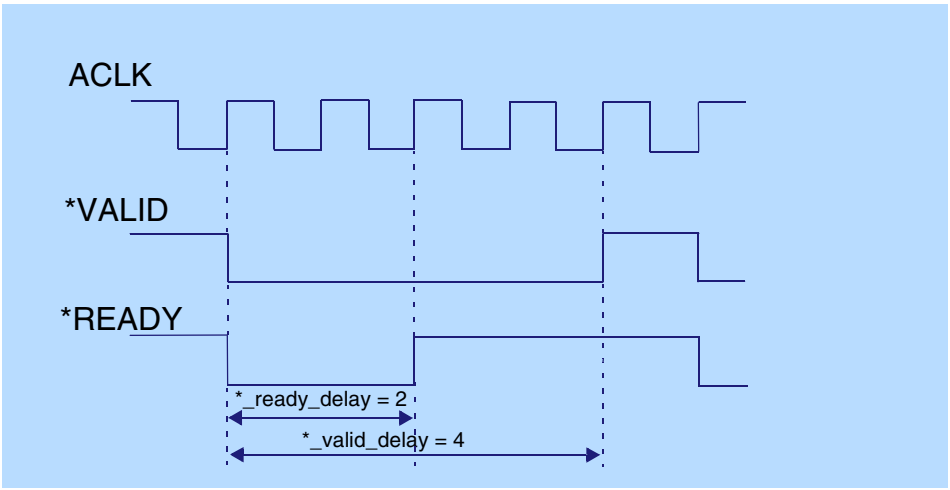
The default configuration (*delay_mode = AXI_VALID2READY*) corresponds to the delay measured from the positive edge of *ACLK* when **VALID* is asserted within a transaction. Figure 1-3 demonstrates how to achieve a **VALID* asserted before **READY* handshake.

Figure 1-3. Operational Transaction Field delay_mode = AXI_VALID2READY



The other configuration (*delay_mode = AXI_TRANS2READY*) corresponds to the delay measured from the completion of a previous transaction phase (**VALID* and **READY* both asserted). Figure 1-4 demonstrates how to achieve a **READY* before **VALID* handshake.

Figure 1-4. Operational Transaction Field delay_mode = AXI_TRANS2READY



Data Beat Done

There is a *data_beat_done* transaction field for each transaction, defined as an array, to indicate when each data phase (beat) has completed. Each element of the *data_beat_done* array is set to 1 when each data phase (beat) has completed within a data burst.

You can call the *get_read_data_phase()* task at the master BFM to investigate how many beats of a read data burst have completed by analyzing how many elements of the *data_beat_done* array have been set to 1. Similarly, the g*et_write_data_phase()* task can be called at the slave BFM to analyze a write data burst.

Transaction Done

The *transaction_done* transaction field in each transaction indicates when the transaction has completed.

You can call the *get_read_data_burst()* task at the master BFM to investigate whether a transaction has completed for a read transfer. For a write transfer, you can call the *get_write_response_phase(),* also at the master BFM, to determine whether a transaction has completed.

AXI3 Protocol Restrictions

There are two features of the AMBA AXI Protocol Specification that are not supported to simplify the API to create the desired protocol stimulus.

The ability to start a write data burst before its write address phase in a transaction is not supported. The ability to start a write data burst simultaneously with its write address phase is however supported. This modifies the AXI3 protocol specification write transaction handshake dependencies diagram, Figure 3-5 in section 3.3, by effectively adding double-headed arrows between *AWVALID* to *WVALID* and *AWREADY* to *WVALID*, with the provision that they can be simultaneous.

The ability to interleave read or write data is not supported. Therefore, a data burst that has started will complete before another data burst with the same or different transaction ID can start. This modifies the AXI3 protocol specification by effectively removing the statement concerning the interleaving of read data with different *ARID* fields in section 8.3. It also completely removes section 8.5 concerning write data interleaving.

All other aspects of the AMBA AXI Protocol Specification, Version 2, are supported.

Simulator GCC Requirements

Mentor Verification IP requires that the simulator's installation directory includes the GCC libraries shown in Table 1-2. If the installation of the GCC libraries was an optional part of the simulator's installation and the Mentor VIP does not find these libraries, you will see an error similar to the following error:

```
ModelSim / Questa SIM
# ** Error: (vsim-8388) Could not find the MVC shared library : GCC not
found in installation directory (/home/user/altera2/11.1sp1-
12.0b92/modelsim_ase) for platform "linux".  Please install GCC version
"gcc-4.3.3-linux"
```

Table 1-2. Simulator GCC Requirements

Simulator	Version	GCC version(s)	Search Path
Mentor Questa SIM /ModelSim			
	10.0d	4.3.3 (Linux 32-bit)	<install dir>/gcc-4.3.3-linux
		4.3.3 (Linux 64-bit)	<install dir>/gcc-4.3.3-linux_x86_64
		4.2.1 (Windows 32-bit)	<install dir>/gcc-4.2.1-mingw32vc9
Synopsys VCS/VCS-MX			
	2011.12	4.2.2 (Linux 32-bit)	$VCS_HOME/gnu/linux/4.2.2_32-shared $VCS_HOME/gnu/4.2.2_32-shared
		4.2.2 (Linux 64-bit)	$VCS_HOME/gnu/linux/4.2.2_64-shared $VCS_HOME/gnu/4.2.2_64-shared
	Note: If the environment variable VG_GNU_PACKAGE is set, then this will be used instead of the VCS_HOME environment variable.		
Cadence Incisiv			
	11.10	4.4 (Linux 32/64-bit)	<install dir>/tools/cdsgcc/gcc/4.4
	Note: Use the cds_tools.sh executable to find the Incisiv installation. Ensure $PATH includes the Installation path and <install dir>/tools/cdsgcc/gcc/4/4/install/bin. Also, ensure the LD_LIBRARY_PATH includes <install dir>/tools/cdsgcc/gcc/4/4/install/lib		

Chapter 2
SystemVerilog AMBA AXI3 BFMs

This section provides information about the AMBA BFMs API for the Mentor Verification IP Altera Edition.

AMBA AXI3 Master BFM

The AMBA AXI3 Master BFM implements the full AMBA AXI3 protocol. In addition to the standard protocol, it supports user sideband signals *AWUSER* and *ARUSER*.

AXI3 Master Timing

For detailed timing diagrams of the protocol bus activity, refer to the AMBAAXI Protocol Specification, which you can use as the reference for details of the following BFM and events.

This specification does not define any timescale or clock period with signal events sampled and driven at rising *ACLK* edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the testbench and design IP as a result of

- ` timescale directives in design elements.
- timeprecision declarations in design elements.
- compiler command-line options.
- simulation command-line options.
- local, or site-wide, simulator initialization files.

If there is no timescale directive, the default time unit and time precision are tool specific.The recommended practice is to use timeunit and timeprecision declarations (Refer to the SystemVerilog LRM section 3.14 for details).

AXI3 Master Configuration

The AMBA AXI3 Master BFM supports the full range of signals defined for the AMBA AXI3 protocol. You may customize the AMBAAXI3 Master interface using the configuration fields described in Table 2-1.

Table 2-1. AXI3 Master BFM Configuration

Configuration Field	Description
Signal Width #Parameters	
AXI_ADDRESS_WIDTH	Address width in bits. Refer to the AMBA AXI Protocol specification for more details. Default: 32.
AXI_RDATA_WIDTH	Read data width in bits. Refer to the AMBA AXI Protocol specification for more details. Default: 1024.
AXI_WDATA_WIDTH	Write data width in bits. Refer to the AMBA AXI Protocol specification for more details. Default: 1024.
AXI_ID_WIDTH	ID width in bits. Refer to the AMBA AXI Protocol specification for more details. Default: 4.
Timing Variables	
AXI_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI_CONFIG_MAX_ TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_ DATA_TRANSFER	The maximum number of write data beats that the AXI interface can generate as part of write data burst of write transfer. Default: 1024.
AXI_CONFIG_BURST_TIMEOUT_ FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
Master Attributes	
AXI_CONFIG_WRITE_CTRL_TO_ DATA_MINTIME	The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_WRITE_DATA_ TO_CTRL_MINTIME	The minimum delay between the start of a write data phase to the start of a write response phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_DEFAULT_ UNDER_RESET	The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true

Table 2-1. AXI3 Master BFM Configuration

Slave Attributes	
AXI_CONFIG_SUPPORT_ EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI protocol specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_ UNDER_RESET	The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol specification for more details. 0 = false (default) 1 = true
Error Detection	
AXI_CONFIG_ENABLE_ERRORS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to AXI3 Master Timing for details of simulator time-steps.

AXI3 Master Assertions

The master BFM performs protocol error checking via built-in assertions.

Note

The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the set_config() command as the following example illustrates:

```
set_config(AXI_CONFIG_ENABLE_ERRORS,0)
```

Alternatively, individual built-in assertions may be disabled by using a sequence of get_config() and set_config() commands on the respective assertion. For example, to disable assertion checking for the *AWLOCK* signal changing between the *AWVALID* and *AWREADY* handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI_CONFIG_ENABLE_ERROR);

// Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI_CONFIG_ENABLE_ERROR, config_assert_bitvector);
```

Note

Do not confuse the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector with the *AXI_CONFIG_ENABLE_ERRORS* global enable/disable.

To re-enable the *AXI_LOCK_CHANGED_BEFORE_AWREADY* assertion, following the above code sequence, assign the assertion within the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector to 1.

For a complete listing of assertions, refer to "AXI3 Assertions" on page 87.

SystemVerilog AXI3 Master Application Program Interface

This section describes the SystemVerilog AMBA AXI3 Master API for the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the BFM.

Table 2-2. SystemVerilog AXI3 Master set_config()

Prototype

```
function void set_config
(
   input axi_config_e config_name,
   input axi_max_bits_t config_val
);
```

Arguments

config_name	config_val
AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "AXI3 Master BFM Configuration" on page 24 for descriptions and valid values.

Returns

None

Usage

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1);
Set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the AXI3 Master BFM

Table 2-3. SystemVerilog AXI3 Master get_config()

Prototype

```
function axi_max_bits_t get_config
(
   input axi_config_e config_name
);
```

Arguments

config_name
AXI_CONFIG_SETUP_TIME
AXI_CONFIG_HOLD_TIME
AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
AXI_CONFIG_BURST_TIMEOUT_FACTOR
AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME
AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET
AXI_CONFIG_ENABLE_ERROR
AXI_CONFIG_ENABLE_ERRORS
AXI_CONFIG_MASTER_ERROR_POSITION
AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

Returns

axi_max_bits_t

Usage

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS);
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_write_transaction()

This function creates a read transaction with a user-supplied start address and burst length, and then returns the transaction. All other transaction fields default to legal protocol values. Any transaction field can then be modified in the returned transaction.

Table 2-4. SystemVerilog AXI3 Master create_write_transaction Function

Prototype

```
function automatic axi_transaction
create_write_transaction
(
    input bit [((AXI_ADDRESS_WIDTH) - 1):0]  addr,
    bit [3:0] burst_length = 0
)
```

Transaction Variables

Arguments	addr	Start address
	burst_length	Burst length (default = 0)
Protocol Transaction Fields	size	Burst size (defaults to width of bus) – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
	burst	Burst type – AXI_FIXED; AXI_INCR (default); AXI_WRAP; AXI_BURST_RSVD;
	lock	Burst lock – AXI_NORMAL (default); AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;

Protocol Transaction Fields (continued)

cache	Burst cache – AXI_NONCACHE_NONBUF (default); AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;
prot	Burst protection – AXI_NORM_SEC_DATA (default); AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
id	Burst ID
data_words[]	Data words array
write_strobes[]	Write strobes array – Each element 0 or 1
resp	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;

Operational Transaction Fields	gen_write_strobes	Generate write strobes flag – 0 = user supplied write strobes 1 = auto-generated write strobes (default)
	operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING (default);
	delay_mode	AXI_VALID2READY (default); AXI_TRANS2READY;
	write_data_mode	AXI_DATA_AFTER_ADDRESS (default); AXI_DATA_WITH_ADDRESS;
	address_valid_delay	Write address channel AWVALID delay in ACLK cycles for this transaction (default = 0)
	data_valid_delay[]	Write data channel WVALID delay array in ACLK cycles for this transaction (default = 0 for all elements)
	write_response_ready_delay	Write response channel BREADY delay in ACLK cycles for this transaction (default = 0)
	data_beat_done[]	Write data channel beat ‘done’ flag array for this transaction
	transaction_done	Write transaction ‘done’ flag for this transaction
Returns	axi_transaction	

Usage

```
// Write data burst length of 3 to start address 16.
trans = bfm.create_write_transaction(16, 3);
    trans.size = AXI_BYTES_4;
    trans.data_words[0] = 'hACE0ACE1;
    trans.data_words[1] = 'hACE2ACE3;
    trans.data_words[2] = 'hACE4ACE5;
    trans.data_words[3] = 'hACE6ACE7;
```

create_read_transaction()

This task executes a master read transaction using the *axi_transaction* record as input in a blocking or nonblocking mode, based on the *operation_mode* configuration. For a write transaction it automatically generates the write strobe values if the user supplied *write_strobes* array is empty. On completion of this transaction, it sets the *transaction_done* flag to 1.

Table 2-5. SystemVerilog AXI3 Master create_read_transaction Function

Prototype

```
function automatic axi_transaction
create_read_transaction
(
    input bit [((AXI_ADDRESS_WIDTH) - 1):0]
addr,
    bit [3:0] burst_length = 0
)
```

Transaction Variables

Arguments	addr	Start address
	burst_length	Burst length (default = 0)
Protocol Transaction Fields	size	Burst size (defaults to width of bus) – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
	burst	Burst type – AXI_FIXED; AXI_INCR (default); AXI_WRAP; AXI_BURST_RSVD;
Protocol Transaction Fields	lock	Burst lock – AXI_NORMAL (default); AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;

Table 2-5. SystemVerilog AXI3 Master create_read_transaction Function

	cache	Burst cache – AXI_NONCACHE_NONBUF (default); AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;
	prot	Burst protection (default) – AXI_NORM_SEC_DATA; AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
	id	Burst ID
	data_words[]	Data words array
	resp[]	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
Operational Transaction Fields	operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING (default);
	delay_mode	Delay mode - AXI_VALID2READY (default); AXI_TRANS2READY;
	address_valid_delay	Read address channel ARVALID delay in ACLK cycles for this transaction (default = 0)

Table 2-5. SystemVerilog AXI3 Master create_read_transaction Function

Operational Transaction Fields	data_ready_delay[]	Read data channel RREADY delay array in ACLK cycles for this transaction (default = 0 for all elements)
	data_beat_done[]	Read data channel beat ‘done’ flag array for this transaction
	transaction_done	Read transaction ‘done’ flag for this transaction
Returns	axi_transaction	

Usage

```
// Read data burst length of 3 to start address 16.
   trans = bfm.create_read_transaction(16, 3);
```

execute_transaction()

This task executes a master transaction using the *axi_transaction* record as input in blocking or nonblocking mode, based on the *operation_mode* configuration. For a write transaction, it auto-corrects the write strobe values if the transaction field *gen_write_strobes = 1*. On completion of a transaction, it sets the *transaction_done* flag to 1.

Table 2-6. SystemVerilog AXI3 Master execute_transaction Task

Prototype	`task automatic execute_transaction(axi_transaction trans)`	
Transaction Variables		
Arguments	trans	axi_transaction record

Usage

```
// By default it will execute a blocking transaction
   bfm.execute_transaction(trans);
```

execute_write_addr_phase()

This task executes a master write address phase with the *axi_transaction* record as input in a blocking or nonblocking mode, based on the *operation_mode* configuration.

Table 2-7. SystemVerilog AXI3 Master execute_write_addr_phase Task

Prototype `task automatic execute_write_addr_phase(axi_transaction trans)`

Transaction Variables

Arguments		
	trans	axi_transaction record

Usage

```
// By default it will execute a blocking transaction
   bfm.execute_write_addr_phase(trans);
```

execute_read_addr_phase()

This task executes a master read address phase with the *axi_transaction* record as input in a blocking or nonblocking mode, based on the *operation_mode* configuration.

Table 2-8. SystemVerilog AXI3 Master execute_read_addr_phase Task

Prototype `task automatic execute_read_addr_phase(axi_transaction trans)`

Transaction Variables

Arguments		
	trans	axi_transaction record

Usage

```
// By default it will execute a blocking transaction
   bfm.execute_read_addr_phase(trans);
```

execute_write_data_burst()

This task executes a master write data burst using the *axi_transaction* record as input in a blocking or nonblocking mode, based on the *operation_mode* configuration. It automatically generates the write data strobes for each data phase (beat) within the burst.

Table 2-9. SystemVerilog AXI3 Master execute_write_data_burst Task

Prototype	`task automatic execute_write_data_burst(axi_transaction trans)`	
Transaction Variables		
Arguments	trans	axi_transaction record

Usage

```
// By default it will execute a blocking transaction
   bfm.execute_write_data_burst(trans);
```

execute_write_data_phase()

This task executes a master write data phase using the *axi_transaction* record and the element index of the data array as input in a blocking or nonblocking mode, based on the *operation_mode* configuration. On completion of the data phase, this task sets the *data_beat_done* flag of the data element to 1.

Table 2-10. SystemVerilog AXI3 Master execute_write_data_phase Task

Prototype

```
task automatic execute_write_data_phase
(
      axi_transaction trans,
      int index = 0
)
```

Transaction Variables

Arguments		
	trans	axi_transaction record

Usage

```
// By default it will execute a blocking transaction
// Execute a data phase using data in element 1 of the data array.
   bfm.execute_write_data_phase(trans, 1);
```

get_read_data_burst()

This task waits for a read data burst and copies the data and response information via the *trans* argument. On completion of the data burst, this task sets the *transaction_done* flag to 1.

Table 2-11. SystemVerilog AXI3 Master get_read_data_burst Task

Prototype

```
task automatic get_read_data_phase
(
    axi_transaction trans,
);
```

Transaction Variables

Arguments	trans	axi_transaction record
Protocol Transaction Fields	last	Last read data beat signal RLAST.
	data_words[]	Read data word array
	resp[]	Read response array
	id	Burst response ID
Operational Transaction Fields	transaction_done	Read transaction ‘done’ flag for this transaction

Usage

```
bfm.get_read_data_burst(read_trans);
```

get_read_data_phase()

This task waits for a read data phase (beat) and copies the data and response information via the '*trans*' argument. It sets the *RREADY* signal value at the appropriate time. On completion of a data phase (beat), this task sets the *data_beat_done[index]* to 1 and sets the *transaction_done* flag to 1 on completion of the burst

Table 2-12. SystemVerilog AXI3 Master get_read_data_phase Task

Prototype

```
task automatic get_read_data_phase
(
    axi_transaction trans,
    int index = 0
);
```

Transaction Variables

Arguments	trans	axi_transaction record
	index	Array index for data beat array element
Protocol Transaction Fields	last	Last read data beat signal RLAST.
	data_words[index]	Read data word array element
	resp[index]	Read response array element
	id	Burst response ID
Operational Transaction Fields	data_beat_done[index]	Read data beat array element *done* flag
	transaction_done	Read transaction 'done' flag for this transaction

Usage

```
bfm.get_read_data_phase(read_trans[i]);
```

get_write_response_phase

This task waits for a write response phase and copies the response information via the *trans* argument. It sets the *BREADY* signal value at the appropriate time. It also sets the *transaction_done* flag to 1 on completion of the write response phase.

Table 2-13. SystemVerilog AXI3 Master get_write_response_phase Task

Prototype	`task automatic get_write_response_phase(axi_transaction trans);`	
Transaction Variables		
Arguments	trans	axi_transaction record
Protocol Transaction Fields	resp	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
	id	Burst response ID
Operational Transaction Fields	transaction_done	Write transaction done flag for this transaction

Usage

```
bfm.get_write_response_phase(write_trans);
```

wait_on()

This tasks waits for an event on the *ACLK* or *ARESETn* signals

Table 2-14. SystemVerilog AXI3 Master wait_on Task

Prototype	`Task automatic wait_on(axi_wait_e phase)`	
Transaction Variables		
Arguments	phase	Wait for - AXI_CLOCK_POSEDGE AXI_CLOCK_NEGEDGE AXI_CLOCK_ANYEDGE AXI_RESET_POSEDGE AXI_RESET_NEGEDGE AXI_RESET_ANYEDGE

Usage

```
bfm.wait_on(AXI_RESET_POSEDGE);
bfm.wait_on(AXI_CLOCK_POSEDGE);
```

Chapter 3
SystemVerilog AMBA AXI3 Slave BFM

The AMBA AXI3 slave BFM implements the full AMBA AXI3 protocol. In addition to the standard protocol it supports user sideband signals AWUSER and ARUSER.

AXI3 Slave Timing

For detailed timing diagrams of the protocol bus activity refer to the AMBA AXI protocol specification to reference the following details of the BFM and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising *ACLK* edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the testbench and design IP as a result of:

- ` timescale directives in design elements.
- timeprecision declarations in design elements.
- compiler command-line options.
- simulation command-line options
- local, or site-wide, simulator initialization files.

If there is no timescale directive, the default time unit and time precision are tool specific.The recommended practice is to use timeunit and timeprecision declarations (Refer to the SystemVerilog LRM section 3.14 for details).

AXI3 Slave Configuration

The AMBA AXI3 slave BFM supports the full range of signals defined for the AMBA AXI3 protocol. You may customize the AMBA AXI3 Slave interface using the configuration fields described in Table 3-1.

Table 3-1. AXI3 Slave BFM Configuration

Configuration Field	Description
Signal Width #Parameter	
AXI_ADDRESS_WIDTH	Address width in bits. Refer to the AMBA AXI protocol specification for more details. Default: 32.
AXI_RDATA_WIDTH	Read data width in bits. Refer to the AMBA AXI protocol specification for more details. Default: 1024.
AXI_WDATA_WIDTH	Write data width in bits. Refer to the AMBA AXI protocol specification for more details. Default: 1024.
AXI_ID_WIDTH	ID width in bits. Refer to the AMBA AXI protocol specification for more details. Default: 4.
Timing Variables	
AXI_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.
AXI_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.
AXI_CONFIG_MAX_ TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_ DATA_TRANSFER	The maximum number of write data beats that the AXI interface can generate as part of write data burst of write transfer. Default: 1024.
AXI_CONFIG_BURST_TIMEOUT_ FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
Master Attributes	
AXI_CONFIG_WRITE_CTRL_TO_ DATA_MINTIME	The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_WRITE_DATA_ TO_CTRL_MINTIME	The minimum delay between the start of a write data phase to the start of a write response phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_DEFAULT_ UNDER_RESET	The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true

Table 3-1. AXI3 Slave BFM Configuration

Slave Attributes	
AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the slave BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the slave BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI protocol specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET	The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI protocol specification for more details. 0 = false (default) 1 = true
Error Detection	
AXI_CONFIG_ENABLE_ERRORS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to AXI3 Slave Timing for details of simulator time-steps.

AXI3 Slave Assertions

The slave BFM performs protocol error checking via built-in assertions.

Note

The built-in BFM assertions are independent of programming language and simulator.

By default all built-in assertions are enabled in the slave BFM. To globally disable them in the slave BFM, use the set_config() command as the example below illustrates:

```
set_config(AXI_CONFIG_ENABLE_ERRORS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of get_config() and set_config() commands on the respective assertion. For example, to disable assertion checking for the *AWLOCK* signal changing between the *AWVALID* and *AWREADY* handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI_CONFIG_ENABLE_ERROR);

// Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI_CONFIG_ENABLE_ERROR, config_assert_bitvector);
```

Note

Do not confuse the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector with the *AXI_CONFIG_ENABLE_ERRORS* global enable/disable.

To re-enable the *AXI_LOCK_CHANGED_BEFORE_AWREADY* assertion, following the above code sequence, assign the assertion within the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector to 1.

For a complete listing of assertions, refer to "AXI3 Assertions" on page 87.

SystemVerilog AXI3 Slave Application Program Interface

This section describes the SystemVerilog AMBA AXI3 Slave API for the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the AXI3 Slave BFM.

Table 3-2. SystemVerilog AXI3 Slave set_config()

Prototype

```
function void set_config
(
   input axi_config_e config_name,
   input axi_max_bits_t config_val
);
```

Arguments

config_name	config_val
AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "AXI3 Slave BFM Configuration" on page 46 for descriptions and valid values.

Returns None

Usage

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1);
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the AXI3 Slave BFM.

Table 3-3. SystemVerilog AXI3 Slave get_config()

Prototype	`function axi_max_bits_t get_config ( input axi_config_e config_name );`
Arguments	**config_name**
	AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS
Returns	axi_max_bits_t

Usage

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS);
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_slave_transaction()

This function creates a slave transaction. All other transaction fields default to legal protocol values, unless assigned a value.

Table 3-4. SystemVerilog AXI3 Slave create_slave_transaction Function

Prototype

```
function automatic axi_transaction
   create_slave_transaction()
```

Transaction Variables

Operational Transaction Fields	delay	Address channel phase A*READY delay in ACLK cycles for this transaction
	delay_mode	Delay mode - AXI_VALID2READY (default); AXI_TRANS2READY;
	operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING (default);
	data_valid_delay[]	Read data channel RVALID delay array in ACLK cycles for this transaction (default = 0 for all elements)
	data_ready_delay[]	Write data channel WREADY delay array in ACLK cycles for this transaction (default = 0 for all elements)
	write_response_valid_delay	Write response channel BVALID delay in ACLK cycles for this transaction (default = 0)
Returns	axi_transaction	

Usage

```
// Create a slave read transaction
read_trans = bfm.create_slave_transaction();
// Create a slave write transaction
write_trans = bfm.create_slave_transaction();
```

execute_read_data_burst()

This task initiates the slave read data burst using the *axi_transaction* record as input. Based on *operation_mode*, this task initiates the transaction in blocking or nonblocking mode. Default operation mode is *AXI_TRANSACTION_BLOCKING*. On completion of the transaction onto the bus, this task sets the *transaction_done* to 1.

Table 3-5. SystemVerilog AXI3 Slave execute_read_data_burst Task

Prototype

```
task automatic execute_read_data_burst
  (axi_transaction trans)
```

Transaction Variables

Arguments	trans	axi_transaction record
Returns	None	

Usage

```
bfm.execute_read_data_burst(read_trans);
```

execute_read_data_phase()

This task initiates the slave read data phase(beat) using the *axi_transaction* record as input. Based on *operation_mode*, this task initiates the transaction in blocking or nonblocking mode. The default operation mode is *AXI_TRANSACTION_BLOCKING*. On completion of one data phase (beat), this task sets the d*ata_beat_done[index]* to 1. This task also sets the *transaction_done* to 1 on completion of the transaction

Table 3-6. SystemVerilog AXI3 Slave execute_read_data_phase Task

Prototype

```
task automatic
  execute_read_data_phase(axi_transaction trans, int
  index = 0);
```

Transaction Variables

Arguments	trans	axi_transaction record

Usage

```
bfm.execute_read_data_phase(read_trans);
```

execute_write_response_phase()

This task initiates the slave write response phase using the *axi_transaction* record as input. Based on *operation_mode*, this task initiates the transaction in blocking or nonblocking mode. The default operation mode is *AXI_TRANSACTION_BLOCKING*. This task sets the *transaction_done* to 1 on completion of a transaction

Table 3-7. SystemVerilog AXI3 Slave execute_write_response_phase Task

Prototype

```
task automatic
  execute_write_response_phase(axi_transaction trans)
```

Transaction Variables

Arguments	trans	axi_transaction record

Usage

```
bfm.execute_write_response_phase(write_trans);
```

get_write_addr_phase()

This task gets a write address phase, copying the information from the *trans* argument.

Table 3-8. SystemVerilog AXI3 Slave execute_write_response_phase Task

Prototype `task automatic get_write_addr_phase(axi_transaction trans)`

Transaction Variables

Arguments	trans	axi_transaction record as input

Usage

```
bfm.get_write_addr_phase(write_trans);
```

get_read_addr_phase()

This task gets a read address phase and copies the address information via the *trans* argument.

Table 3-9. SystemVerilog AXI3 Slave get_write_addr_phase Task

Prototype	`task automatic get_read_addr_phase(axi_transaction trans);`	
Transaction Variables		
Arguments	trans	axi_transaction record

Usage

```
bfm.get_write_addr_phase(write_trans);
```

get_write_data_phase()

This task gets a write data phase (beat) and copies the data information via the *trans* argument. On completion of a single data phase, it sets the *data_beat_done[index]* to 1.

Table 3-10. SystemVerilog AXI3 Slave get_write_data_phase Task

Prototype	`task automatic get_write_data_phase(axi_transaction trans);`	
Transaction Variables		
Arguments	trans	axi_transaction record

Usage

```
bfm.get_write_data_phase(write_trans);
```

get_write_data_burst()

This task gets a write data burst and copies the data information via the *trans* argument.

Table 3-11. SystemVerilog AXI3 Slave get_write_data_burst Task

Prototype

```
task automatic
   get_write_data_burst(axi_transaction trans);
```

Transaction Variables

Arguments	trans	axi_transaction record

Usage

```
bfm.get_write_data_burst(write_trans);
```

wait_on()

This tasks waits for an event on the *ACLK* or *ARESETn* signals.

Table 3-12. SystemVerilog AXI3 Slave wait_on Task

Prototype

```
task automatic wait_on(axi_wait_e phase);
```

Transaction Variables

Arguments

phase	Wait for - AXI_CLOCK_POSEDGE AXI_CLOCK_NEGEDGE AXI_CLOCK_ANYEDGE AXI_RESET_POSEDGE AXI_RESET_NEGEDGE AXI_RESET_ANYEDGE

Usage

```
bfm.wait_on(AXI_RESET_POSEDGE);
bfm.wait_on(AXI_CLOCK_POSEDGE);
```

Helper Functions

AMBA AXI protocols typically provide a start address only within a transaction, with the following addresses for each byte of a data burst calculated using the size, length, and type transaction fields of the transaction. Helper functions are available to provide you with a simple interface to set and get actual address/data values.

get_write_addr_data()

This helper function takes *axi_transaction*(write type) and *index* as inputs. It provides the address and data of the memory for the *index number*. If the corresponding *index* does not exist then this function returns *false*, otherwise it returns *true*.

Table 3-13. SystemVerilog AXI3 Slave get_write_addr_data Function

Prototype

```
function bit get_write_addr_data
(
    input axi_transaction trans,
    input int index = 0,
   output bit [((AXI_ADDRESS_WIDTH) - 1) : 0] addr[],
    output bit [7:0] data[]
);
```

Arguments		
	trans	axi_transaction record
	index	Array element number
	addr[ ]	Write address array
	data[ ]	Write data array
Returns	bit	

Usage

```
bfm.get_write_addr_data(write_trans, i, addr, data);
```

get_read_addr()

This helper function takes *axi_transaction*(read type) and *index* as inputs. It provides the address of the memory for the *index number*. If the corresponding *index* does not exist then this function returns *false*, otherwise it returns *true*.

Table 3-14. SystemVerilog AXI3 Slave get_write_addr Function

Prototype

```
function bit get_read_addr
(
    input axi_transaction trans,
    input int index = 0,
   output bit [((AXI_ADDRESS_WIDTH) - 1) : 0]
addr[]
);
```

Arguments

trans	axi_transaction record
index	Array element number
addr[ ]	Write address array

Returns

bit

Usage

```
bfm.get_read_addr(read_trans, i, addr);
```

set_read_data()

Table 3-15. SystemVerilog AXI3 Slave get_write_addr_data Function

Prototype

```
function void set_read_data
(
    input axi_transaction trans,
    input int index = 0,
    input bit [((AXI_ADDRESS_WIDTH) - 1) : 0] addr[],
    input bit [7:0] data[]
);
```

Arguments

trans	axi_transaction record
index	Array element number
addr[ ]	Read address array
data[ ]	Read data array

Returns

None

Usage

```
set_read_data(read_trans, i, addr, mem_data);
```

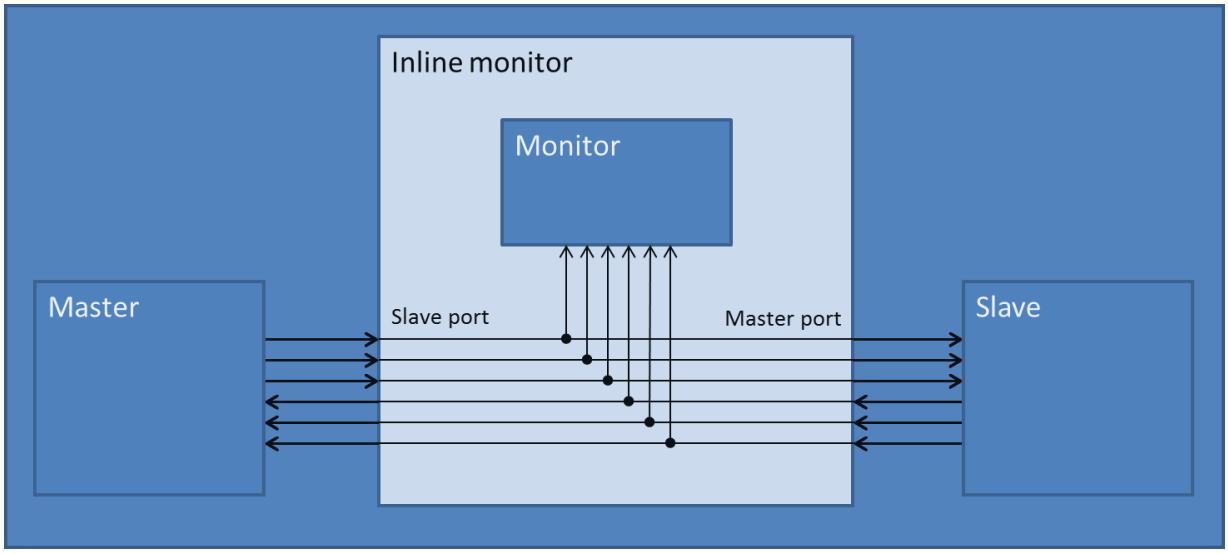
Chapter 4
SystemVerilog AMBA AXI3 Monitor BFM

The AMBA AXI3 monitor BFM implements the full AMBA AXI3 protocol. In addition to the standard protocol it supports user sideband signals *AWUSER* and *ARUSER*.

AXI3 Inline Monitor Connection

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped within an inline monitor interface and connected inline between a master and slave, as shown in Figure 4-1. It has separate master and slave ports and monitors protocol traffic between a master and slave. The monitor itself then has access to all the facilities provided by the monitor BFM.

Figure 4-1. Inline Monitor Connection Diagram



AXI3 Monitor Timing

For detailed timing diagrams of the protocol bus activity refer to the AMBA AXI protocol specification, which you can use as the reference for the following details of the BFM and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising *ACLK* edges. Therefore, the monitor BFM does not contain any timescale,

timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the testbench and design IP as a result of:

- ` timescale directives in design elements.
- timeprecision declarations in design elements.
- compiler command-line options.
- simulation command-line options.
- local, or site-wide, simulator initialization files.

If there is no timescale directive, the default time unit and time precision are tool specific.The recommended practice is to use timeunit and timeprecision declarations (Refer to the SystemVerilog LRM section 3.14 for details).

AXI3 Monitor Configuration

The AMBA® AXI3™ Monitor BFM supports the full range of signals defined for the AMBA AXI3 protocol. You can customize the AMBA AXI3 Monitor interface using the configuration fields described in Table 4-1.

Table 4-1. AXI3 Monitor BFM Configuration

Configuration Field	Description
Signal Width #Parameters	
AXI_ADDRESS_WIDTH	Address width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
AXI_RDATA_WIDTH	Read data width in bits. Refer to the AMBA AXI protocol specification for more details. Default: 1024.
AXI_WDATA_WIDTH	Write data width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 1024.
AXI_ID_WIDTH	ID width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
Timing Variables	
AXI_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
AXI_CONFIG_MAX_ TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.

Table 4-1. AXI3 Monitor BFM Configuration

AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER	The maximum number of write data beats that the AXI interface can generate as part of write data burst of write transfer. Default: 1024.
AXI_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
Master Attributes	
AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME	The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME	The minimum delay between the start of a write data phase to the start of a write response phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET	The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true
Slave Attributes	
AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the slave BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the slave BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI protocol specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET	The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI protocol specification for more details. 0 = false (default) 1 = true
Error Detection	
AXI_CONFIG_ENABLE_ERRORS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to AXI3 Monitor Timing for details of simulator time-steps.

AXI3 Monitor Assertions

The monitor BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the set_config() command as the example below illustrates:

set_config(AXI_CONFIG_ENABLE_ERRORS,0)

Alternatively, you can disable individual built-in assertions by using a sequence of get_config() and set_config() commands on the respective assertion. For example, to disable assertion checking for the *AWLOCK* signal changing between the *AWVALID* and *AWREADY* handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI_CONFIG_ENABLE_ERROR);

// Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI_CONFIG_ENABLE_ERROR, config_assert_bitvector);
```

Note
Do not confuse the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector with the *AXI_CONFIG_ENABLE_ERRORS* global enable/disable.

To re-enable the *AXI_LOCK_CHANGED_BEFORE_AWREADY* assertion, following the above code sequence, assign the assertion within the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector to 1.

For a complete listing of assertions, refer to "AXI3 Assertions" on page 87.

SystemVerilog AXI3 Monitor Application Program Interface

This section describes the SystemVerilog AMBA® AXI3™ Monitor API for the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the AXI3 Monitor BFM.

Table 4-2. SystemVerilog AXI3 Monitor set_config()

Prototype

```
function void set_config
(
   input axi_config_e config_name,
   input axi_max_bits_t config_val
);
```

Arguments	**config_name**	**config_val**
	AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE__ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "AXI3 Monitor BFM Configuration" on page 64 for descriptions and valid values.
Returns	None	

Usage

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1);
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the AXI3 Monitor BFM.

Table 4-3. SystemVerilog AXI3 Monitor get_config()

Prototype

```
function axi_max_bits_t get_config
(
   input axi_config_e config_name
);
```

Arguments

config_name
AXI_CONFIG_SETUP_TIME
AXI_CONFIG_HOLD_TIME
AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
AXI_CONFIG_BURST_TIMEOUT_FACTOR
AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME
AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET
AXI_CONFIG_ENABLE_ERROR
AXI_CONFIG_ENABLE_ERRORS
AXI_CONFIG_MASTER_ERROR_POSITION
AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

Returns

axi_max_bits_t

Usage

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS);
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_monitor_transaction()

This function creates a transaction to monitor protocol transfers.

Table 4-4. SystemVerilog AXI3 Monitor create_monitor_transaction Function

Prototype	`function automatic axi_transaction create_monitor_transaction()`
Transaction Variables	
Returns	axi_transaction

Usage

```
// Create a monitor transaction
trans = bfm.create_monitor_transaction();
```

get_rw_transaction()

This task waits for a complete read/write transaction and then gets the complete transaction content.

Table 4-5. SystemVerilog AXI3 Monitor get_rw_transaction Task

Prototype

```
function automatic axi_transaction
  create_monitor_transaction();
```

Arguments	trans	axi_transaction record
Protocol Transaction Fields	addr	Start address
	burst_length	Burst length
	size	Burst size – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
	burst	Burst type – AXI_FIXED; AXI_INCR; AXI_WRAP; AXI_BURST_RSVD;

Table 4-5. SystemVerilog AXI3 Monitor get_rw_transaction Task

Protocol Transaction Fields	lock	Burst lock – AXI_NORMAL; AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;
	cache	Burst cache – AXI_NONCACHE_NONBUF; AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;
	prot	Burst protection – AXI_NORM_SEC_DATA; AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
	id	Burst ID
	data_words[]	Data words array
	write_strobes[]	Write strobes array – Each element 0 or 1
	resp	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
	addr_user	Address channel user data
	read_or_write	Read or write flag – AXI_TRANS_READ; AXI_TRANS_WRITE;

Table 4-5. SystemVerilog AXI3 Monitor get_rw_transaction Task

Operational Transaction Fields

gen_write_strobes	Generate write strobes flag – 0 = user supplied write strobes 1 = auto-generated write strobes
operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING;
delay_mode	AXI_VALID2READY; AXI_TRANS2READY;
write_data_mode	AXI_DATA_AFTER_ADDRESS; AXI_DATA_WITH_ADDRESS;
address_valid_delay	Write address channel AWVALID delay in ACLK cycles for this transaction
data_valid_delay[]	Write data channel WVALID delay array in ACLK cycles for this transaction
write_response_ ready_delay	Write response channel BREADY delay in ACLK cycles for this transaction
data_beat_done[]	Write data channel beat ‘done’ flag array for this transaction
transaction_done	Write transaction ‘done’ flag for this transaction

Usage

```
bfm.get_rw_transaction(trans);
```

get_write_addr_phase()

This task waits for a write address phase and then gets write address phase content via the *trans* argument.

Table 4-6. SystemVerilog AXI3 Monitor get_write_addr_phase Task

Prototype

```
task automatic
   get_write_addr_phase(axi_transaction trans)
```

Transaction Variables

Arguments	trans	axi_transaction record as input
Protocol Transaction Fields	addr	Start address
	burst_length	Burst length
	size	Burst size – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
Protocol Transaction Fields	burst	Burst type – AXI_FIXED; AXI_INCR; AXI_WRAP; AXI_BURST_RSVD;
	lock	Burst lock – AXI_NORMAL; AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;

Table 4-6. SystemVerilog AXI3 Monitor get_write_addr_phase Task

cache	Burst cache – AXI_NONCACHE_NONBUF; AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;
prot	Burst protection – AXI_NORM_SEC_DATA; AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
id	Burst ID
addr_user	Address channel user data

Usage

```
bfm.get_write_addr_phase(write_trans);
```

get_read_addr_phase()

This task waits for a read address phase and then gets the read address content via the *trans* argument.

Table 4-7. SystemVerilog AXI3 Monitor get_write_addr_phase Task

Prototype

```
task automatic
  get_read_addr_phase(axi_transaction trans)
```

Transaction Variables

Arguments	trans	axi_transaction record
Protocol Transaction Fields	addr	Start address

Table 4-7. SystemVerilog AXI3 Monitor get_write_addr_phase Task

burst_length	Burst length
size	Burst size – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
burst	Burst type – AXI_FIXED; AXI_INCR; AXI_WRAP; AXI_BURST_RSVD;
lock	Burst lock – AXI_NORMAL; AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;
cache	Burst cache – AXI_NONCACHE_NONBUF; AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;

Table 4-7. SystemVerilog AXI3 Monitor get_write_addr_phase Task

Protocol Transaction Fields	prot	Burst protection – AXI_NORM_SEC_DATA; AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
	id	Burst ID
	addr_user	Address channel user data

Usage

```
bfm.get_read_addr_phase(read_trans);
```

get_write_data_phase()

This task waits for a write data phase (beat) and then gets the write data beat content via the *trans* argument. On completion of one data phase(beat), this task sets the *data_beat_done[index]* to 1.

Table 4-8. SystemVerilog AXI3 Monitor get_write_data_phase Task

Prototype

```
task automatic
  get_write_data_phase(axi_transaction trans)
```

Transaction Variables

Arguments	trans	axi_transaction record
Protocol Transaction Fields	last	Last write data beat signal WLAST.
	data_words[index]	Write data word array element
	id	Write data beat ID
Operational Transaction Fields	data_beat_done[index]	Write data beat array element ‘done’ flag

Usage

```
bfm.get_write_data_phase(write_trans);
```

get_write_data_burst()

This task waits for a write data burst and copies the data burst information via the *trans* argument.

Table 4-9. SystemVerilog AXI3 Monitor get_write_data_burst Task

Prototype

```
task automatic
   get_write_data_burst(axi_transaction trans)
```

Transaction Variables

Arguments	trans	axi_transaction record
Protocol Transaction Fields	id	Write data burst ID
	data_words[]	Write data words array
	write_strobes[]	Write strobes array – Each element 0 or 1

Usage

```
bfm.get_write_data_burst(write_trans);
```

get_write_response_phase

This task waits for a write response and then gets the response phase content via the *trans* argument. This task sets the *transaction_done* to 1 on completion of write response phase.

Table 4-10. SystemVerilog AXI3 Monitor get_write_response_phase Task

Prototype

```
task automatic
   get_write_response_phase(axi_transaction
   trans);
```

Transaction Variables

Arguments	trans	axi_transaction record
Protocol Transaction Fields	resp	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
	id	Burst data ID
Operational Transaction Fields	transaction_done	Write transaction ‘done’ flag for this transaction

Usage

```
bfm.get_write_response_phase(write_trans);
```

get_read_data_phase()

This task waits for a read data phase (beat) and then gets the read data beat content via the trans argument. On completion of one data phase(beat), this task sets the *data_beat_done[index]* to 1.

This task also sets the *transaction_done* to 1 on completion of transaction.

Table 4-11. SystemVerilog AXI3 Monitor get_read_data_phase Task

Prototype

```
task automatic
  get_read_data_phase(axi_transaction trans)
```

Transaction Variables

Arguments	trans	axi_transaction record
Protocol Transaction Fields	last	Last read data beat signal RLAST.
	data_words[index]	Read data word array element
	resp	Read data beat response
	id	Read data beat ID
Operational Transaction Fields	data_beat_done[index]	Read data beat array element ‘done’ flag

Usage

```
bfm.get_read_data_phase(read_trans);
```

get_read_data_burst()

This task waits for a read data burst and then gets the read burst content via the trans argument. On completion of a transaction onto the bus, this task sets the *transaction_done* to 1.

Table 4-12. SystemVerilog AXI3 Monitor get_read_data_burst Task

Prototype	`task automatic get_read_data_burst(axi_transaction trans)`	
Transaction Variables		
Arguments	trans	axi_transaction record
	data_words[]	Read data words array
	id	Read data burst ID
	resp[]	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
Operational Transaction Fields	transaction_done	Read transaction ‘done’ flag for this transaction

Usage

```
bfm.get_read_data_burst(read_trans);
```

wait_on()

This task waits for particular event from the BFM.

Table 4-13. SystemVerilog AXI3 Monitor wait_on ()

Prototype	`task automatic wait_on(axi_wait_e phase)`
Arguments	phase AXI_CLOCK_POSEDGE AXI_CLOCK_NEGEDGE AXI_CLOCK_ANYEDGE AXI_RESET_POSEDGE AXI_RESET_NEGEDGE AXI_RESET_ANYEDGE
Returns	None

Usage

```
bfm.wait_on(AXI_RESET_POSEDGE);
bfm.wait_on(AXI_CLOCK_POSEDGE);
```

Helper Functions

AMBA AXI protocols typically provide a start address only within a transaction, with the following addresses for each byte of a data burst calculated using the size, length, and type transaction fields. Helper functions are available to provide you with a simple interface to set and get actual address/data values.

get_write_addr_data()

This helper function takes *axi_transaction* (write type) and *index* as inputs. It provides the address and data of the memory for the *index* number. If the corresponding *index* does not exist, then this function returns *false*, otherwise it returns *true*.

Table 4-14. SystemVerilog AXI3 Monitor get_write_addr_data Function

Prototype

```
function bit get_write_addr_data
(
    input axi_transaction trans,
    input int index = 0,
    output bit [((AXI_ADDRESS_WIDTH) - 1) : 0]
addr[],
    output bit [7:0] data[]
);
```

Arguments		
	trans	axi_transaction record
	index	Array element number
	addr[ ]	Write address array
	data[ ]	Write data array
Returns	bit	

Usage

```
bfm.get_write_addr_data(write_trans, i, addr, data);
```

get_read_addr()

This helper function takes *axi_transaction* (read type) and index as inputs. It provides the address of the memory for the *index* number. If the corresponding *index* does not exist then this function returns *false*, otherwise it returns *true*.

Table 4-15. SystemVerilog AXI3 Monitor get_read_addr Function

Prototype

```
function bit get_read_addr
(
    input axi_transaction trans,
    input int index = 0,
    output bit [((AXI_ADDRESS_WIDTH) - 1) : 0] addr[]
);
```

Arguments

trans	axi_transaction record
index	Array element number
addr[ ]	Read address array

Returns

bit

Usage

```
bfm.get_read_addr(read_trans, i, addr);
```

set_read_data()

This helper function takes *axi_transaction* (read type), array index, plus the address and data arrays as inputs. It then fills the data array using the *index* number.

Table 4-16. SystemVerilog AXI3 Monitor set_read_data Function

Prototype

```
function void set_read_data
(
    input axi_transaction trans,
    input int index = 0,
    input bit [((AXI_ADDRESS_WIDTH) - 1) : 0] addr[],
    input bit [7:0] data[]
);
```

Arguments

trans	axi_transaction record
index	Array element number
addr[ ]	Read address array
data[ ]	Read data array

Returns

None

Usage

```
bfm.set_read_data(read_trans, i, addr, mem_data);
```

Chapter 5
AXI3 Assertions

The AXI3 Master, Slave, and Monitor BFMs all support error checking via the firing of one or more assertions when a property detailed in the AMBA AXI Protocol Specification has been violated. Each assertion may be individually enabled/disabled using the *set_config()* function for a particular BFM. The property covered for each assertion is noted in Table 5-1 under the Property Reference column. The reference number refers to the section number in the AMBA AXI Protocol Specification.

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60000	AXI_ADDR_FOR_READ_BURST_ACROSS_4K_BOUNDARY	This read transaction has crossed a 4KB boundary.	4.1
AXI3-60001	AXI_ADDR_FOR_WRITE_BURST_ACROSS_4K_BOUNDARY	This write transaction has crossed a 4KB boundary.	4.1
AXI3-60002	AXI_ARADDR_CHANGED_BEFORE_ARREADY	The value of ARADDR has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60003	AXI_ARADDR_UNKN	ARADDR has an X or Z value.	2.5
AXI3-60004	AXI_ARBURST_CHANGED_BEFORE_ARREADY	The value of ARBURST has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60005	AXI_ARBURST_UNKN	ARBURST has an X or Z value.	2.5
AXI3-60006	AXI_ARCACHE_CHANGED_BEFORE_ARREADY	The value of ARCACHE has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60007	AXI_ARCACHE_UNKN	ARCACHE has an X or Z value.	2.5

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60008	AXI_ARID_CHANGED_BEFORE_ARREADY	The value of ARID has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60009	AXI_ARID_UNKN	ARID has an X or Z value.	2.5
AXI3-60010	AXI_ARLEN_CHANGED_BEFORE_ARREADY	The value of ARLEN has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60011	AXI_ARLEN_UNKN	ARLEN has an X or Z value.	2.5
AXI3-60012	AXI_ARLOCK_CHANGED_BEFORE_ARREADY	The value of ARLOCK has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60013	AXI_ARLOCK_UNKN	ARLOCK has an X or Z value.	2.5
AXI3-60014	AXI_ARPROT_CHANGED_BEFORE_ARREADY	The value of ARPROT has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60015	AXI_ARPROT_UNKN	ARPROT has an X or Z value.	2.5
AXI3-60016	AXI_ARREADY_UNKN	ARREADY has an X or Z value.	2.5
AXI3-60017	AXI_ARSIZE_CHANGED_BEFORE_ARREADY	The value of ARSIZE has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60018	AXI_ARSIZE_UNKN	ARSIZE has an X or Z value.	2.5
AXI3-60019	AXI_ARUSER_CHANGED_BEFORE_ARREADY	The value of ARUSER has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	3.1
AXI3-60020	AXI_ARUSER_UNKN	ARUSER has an X or Z value.	2.5

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60021	AXI_ARVALID_DEASSERTED_BEFORE_ARREADY	ARVALID has been de-asserted before ARREADY was asserted.	3.1
AXI3-60022	AXI_ARVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A master interface must begin driving ARVALID high only at a rising clock edge after ARESETn is HIGH.	11.1.2
AXI3-60023	AXI_ARVALID_UNKN	ARVALID has an X or Z value.	2.5
AXI3-60024	AXI_AWADDR_CHANGED_BEFORE_AWREADY	The value of AWADDR has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1
AXI3-60025	AXI_AWADDR_UNKN	AWADDR has an X or Z value.	2.2
AXI3-60026	AXI_AWBURST_CHANGED_BEFORE_AWREADY	The value of AWBURST has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1
AXI3-60027	AXI_AWBURST_UNKN	AWBURST has an X or Z value.	2.2
AXI3-60028	AXI_AWCACHE_CHANGED_BEFORE_AWREADY	The value of AWCACHE has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1
AXI3-60029	AXI_AWCACHE_UNKN	AWCACHE has an X or Z value.	2.2
AXI3-60030	AXI_AWID_CHANGED_BEFORE_AWREADY	The value of AWID has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted (SPEC3(3.1))	3.1
AXI3-60031	AXI_AWID_UNKN	AWID has an X or Z value.	2.2
AXI3-60032	AXI_AWLEN_CHANGED_BEFORE_AWREADY	The value of AWLEN has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1
AXI3-60033	AXI_AWLEN_UNKN	AWLEN has an X or Z value.	2.2
AXI3-60034	AXI_AWLOCK_CHANGED_BEFORE_AWREADY	The value of AWLOCK has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60035	AXI_AWLOCK_UNKN	AWLOCK has an X or Z value.	2.2
AXI3-60036	AXI_AWPROT_CHANGED_BEFORE_AWREADY	The value of AWPROT has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1
AXI3-60037	AXI_AWPROT_UNKN	AWPROT has an X or Z value.	2.2
AXI3-60038	AXI_AWREADY_UNKN	AWREADY has an X or Z value.	2.2
AXI3-60039	AXI_AWSIZE_CHANGED_BEFORE_AWREADY	The value of AWSIZE has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1
AXI3-60040	AXI_AWSIZE_UNKN	AWSIZE has an X or Z value.	2.2
AXI3-60041	AXI_AWUSER_CHANGED_BEFORE_AWREADY	The value of AWUSER has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	3.1
AXI3-60042	AXI_AWUSER_UNKN	AWUSER has an X or Z value.	2.2
AXI3-60043	AXI_AWVALID_DEASSERTED_BEFORE_AWREADY	AWVALID has been de-asserted before AWREADY was asserted.	3.1
AXI3-60044	AXI_AWVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A master interface must begin driving AWVALID high only at a rising clock edge after ARESETn is HIGH.	11.1.2
AXI3-60045	AXI_AWVALID_UNKN	AWVALID has an X or Z value.	2.2
AXI3-60046	AXI_BID_CHANGED_BEFORE_BREADY	The value of BID has changed from its initial value between the time BVALID was asserted, and before BREADY was asserted.	3.1
AXI3-60047	AXI_BID_UNKN	BID has an X or Z value.	2.4
AXI3-60048	AXI_BREADY_NOT_ASSERTED_AFTER_BVALID	Once BVALID has been asserted, BREADY should be asserted within config_max_latency_BVALID_assertion_to_BREADY clock periods.	-
AXI3-60049	AXI_BREADY_UNKN	BREADY has an X or Z value.	2.4

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60050	AXI_BRESP_CHANGED_BEFORE_BREADY	The value of BRESP has changed from its initial value between the time BVALID was asserted, and before BREADY was asserted (SPEC3(3.1)).	3.1
AXI3-60051	AXI_BRESP_UNKN	BRESP has an X or Z value.	2.4
AXI3-60054	AXI_BVALID_DEASSERTED_BEFORE_BREADY	BVALID has been de-asserted before BREADY was asserted.	3.1
AXI3-60055	AXI_BVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A slave interface must begin driving BVALID high only at a rising clock edge after ARESETn is HIGH.	11.1.2
AXI3-60056	AXI_BVALID_UNKN	BVALID has an X or Z value.	2.4
AXI3-60057	AXI_EXCLUSIVE_READ_ACCESS_MODIFIABLE	The modifiable bit (bit 1 of the cache transaction field) should not be set for an exclusive read access.	6.2.4
AXI3-60058	AXI_EXCLUSIVE_READ_BYTES_TRANSFER_EXCEEDS_128	Number of bytes in an exclusive read transaction must be less than or equal to 128.	6.2.4
AXI3-60059	AXI_EXCLUSIVE_WRITE_BYTES_TRANSFER_EXCEEDS_128	Number of bytes in an exclusive write transaction must be less than or equal to 128.	6.2.4
AXI3-60060	AXI_EXCLUSIVE_READ_BYTES_TRANSFER_NOT_POWER_OF_2	Number of bytes of an exclusive read transaction is not a power of 2.	6.2.4
AXI3-60061	AXI_EXCLUSIVE_WRITE_BYTES_TRANSFER_NOT_POWER_OF_2	Number of bytes of an exclusive write transaction is not a power of 2.	6.2.4
AXI3-60062	AXI_EXCLUSIVE_WR_ADDRESS_NOT_SAME_AS_RD	Exclusive write does not match the address of the previous exclusive read to this id.	6.2.4
AXI3-60063	AXI_EXCLUSIVE_WR_BURST_NOT_SAME_AS_RD	Exclusive write does not match the burst setting of the previous exclusive read to this id.	6.2.4
AXI3-60064	AXI_EXCLUSIVE_WR_CACHE_NOT_SAME_AS_RD	Exclusive write does not match the cache setting of the previous exclusive read to this id.	6.2.4

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60065	AXI_EXCLUSIVE_WRITE_ACCESS_MODIFIABLE	The modifiable bit (bit 1 of the cache transaction field) should not be set for an exclusive write access.	6.2.4
AXI3-60066	AXI_EXCLUSIVE_WR_LENGTH_NOT_SAME_AS_RD	Exclusive write does not match the length of the previous exclusive read to this id.	6.2.4
AXI3-60067	AXI_EXCLUSIVE_WR_PROT_NOT_SAME_AS_RD	Exclusive write does not match the prot setting of the previous exclusive read to this id.	6.2.4
AXI3-60068	AXI_EXCLUSIVE_WR_SIZE_NOT_SAME_AS_RD	Exclusive write does not match the size of the previous exclusive read to this id.	6.2.4
AXI3-60069	AXI_EXOKAY_RESPONSE_NORMAL_READ	Slave has responded AXI_EXOKAY to a non-exclusive read transfer.	6.2.4
AXI3-60070	AXI_EXOKAY_RESPONSE_NORMAL_WRITE	Slave has responded AXI_EXOKAY to a non-exclusive write transfer.	6.2.4
AXI3-60071	AXI_EX_RD_RESP_MISMATCHED_WITH_EXPECTED_RESP	Expected response to this exclusive read did not matched with the actual response.	6.2.4
AXI3-60072	AXI_EX_WR_RESP_MISMATCHED_WITH_EXPECTED_RESP	Expected response to this exclusive write did not matched with the actual response.	6.2.4
AXI3-60073	AXI_EX_RD_EXOKAY_RESP_SLAVE_WITHOUT_EXCLUSIVE_ACCESS	Response for an exclusive read to a slave which does not support exclusive access should be AXI_OKAY, but it returned AXI_EXOKAY.	6.2.5
AXI3-60074	AXI_EX_WRITE_BEFORE_EX_READ_RESPONSE	Exclusive write has occurred, with no previous exclusive read.	6.2.4
AXI3-60075	AXI_EX_WRITE_EXOKAY_RESP_SLAVE_WITHOUT_EXCLUSIVE_ACCESS	Response for an exclusive write to a slave which does not support exclusive access should be AXI_OKAY, but it returned AXI_EXOKAY.	6.2.5
AXI3-60076	AXI_ILLEGAL_LENGTH_WRAPPING_READ_BURST	In the last read address phase burst_length has an illegal value for a burst of type AXI_WRAP.	4.4.3
AXI3-60077	AXI_ILLEGAL_LENGTH_WRAPPING_WRITE_BURST	In the last write address phase burst_length has an illegal value for a burst of type AXI_WRAP.	4.4.3
AXI3-60078	AXI_ILLEGAL_RESPONSE_EXCLUSIVE_READ	Response for an exclusive read should be either AXI_OKAY or AXI_EXOKAY.	6.2.4
AXI3-60079	AXI_ILLEGAL_RESPONSE_EXCLUSIVE_WRITE	Response for an exclusive write should be either AXI_OKAY or AXI_EXOKAY.	6.2.4
AXI3-60080	AXI_PARAM_READ_DATA_BUS_WIDTH	The value of AXI_RDATA_WIDTH must be one of 8,16,32,64,128,256,512,1024.	1.2.1

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60081	AXI_PARAM_WRITE_DATA_BUS_WIDTH	The value of AXI_WDATA_WIDTH must be one of 8,16,32,64,128,256,512,1024.	1.2.1
AXI3-60082	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_12	The RA bit of the cache transaction field should not be HIGH when the Modifiable bit is LOW.	5.1
AXI3-60083	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_13	The RA bit of the cache transaction field should not be HIGH when the Modifiable bit is LOW.	5.1
AXI3-60084	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_4	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1
AXI3-60085	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_5	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1
AXI3-60086	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_8	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1
AXI3-60087	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_9	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1
AXI3-60088	AXI_READ_BURST_LENGTH_VIOLATION	The number of beats actually read does not match the burst length defined by the ARLEN.	4.2
AXI3-60089	AXI_READ_BURST_SIZE_VIOLATION	In this read transaction, size has been set greater than the defined data bus.	4.3
AXI3-60090	AXI_READ_DATA_BEFORE_ADDRESS	An unexpected read response has occurred (there are no outstanding read transactions with this id).	3.3
AXI3-60091	AXI_READ_DATA_CHANGED_BEFORE_RREADY	The value of RDATA has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	3.1
AXI3-60092	AXI_READ_DATA_UNKN	RDATA has an X or Z value.	2.6
AXI3-60093	AXI_RESERVED_ARLOCK_ENCODING	The reserved encoding of 2'b11 should not be used for ARLOCK.	6.1
AXI3-60094	AXI_READ_RESP_CHANGED_BEFORE_RREADY	The value of RRESP has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	3.1

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60095	AXI_RESERVED_ARBURST_ENCODING	The reserved encoding of 2'b11 should not be used for ARBURST.	4.4
AXI3-60096	AXI_RESERVED_AWBURST_ENCODING	The reserved encoding of 2'b11 should not be used for AWBURST.	4.4
AXI3-60097	AXI_RID_CHANGED_BEFORE_RREADY	The value of RID has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	3.1
AXI3-60098	AXI_RID_UNKN	RID has an X or Z value.	2.6
AXI3-60099	AXI_RLAST_CHANGED_BEFORE_RREADY	The value of RLAST has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	3.1
AXI3-60100	AXI_RLAST_UNKN	RLAST has an X or Z value.	2.6
AXI3-60101	AXI_RREADY_UNKN	RREADY has an X or Z value.	2.6
AXI3-60102	AXI_RRESP_UNKN	RRESP has an X or Z value.	2.6
AXI3-60105	AXI_RVALID_DEASSERTED_BEFORE_RREADY	RVALID has been de-asserted before RREADY was asserted.	3.1
AXI3-60106	AXI_RVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A slave interface must begin driving RVALID high only at a rising clock edge after ARESETn is HIGH.	11.1.2
AXI3-60107	AXI_RVALID_UNKN	RVALID has an X or Z value.	2.6
AXI3-60108	AXI_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_READ	Exclusive read accesses must have address aligned to the total number of bytes in the transaction.	6.2.4
AXI3-60109	AXI_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_WRITE	Exclusive write accesses must have address aligned to the total number of bytes in the transaction.	6.2.4
AXI3-60110	AXI_UNALIGNED_ADDR_FOR_WRAPPING_READ_BURST	Wrapping bursts must have address aligned to the start of the read transfer.	4.4.3
AXI3-60111	AXI_UNALIGNED_ADDR_FOR_WRAPPING_WRITE_BURST	Wrapping bursts must have address aligned to the start of the write transfer.	4.4.3

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60112	AXI_WDATA_CHANGED_BEFORE_WREADY_ON_INVALID_LANE	On a lane whose strobe is 0, the value of WDATA has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	3.1
AXI3-60113	AXI_WDATA_CHANGED_BEFORE_WREADY_ON_VALID_LANE	On a lane whose strobe is 1, the value of WDATA has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	3.1
AXI3-60114	AXI_WLAST_CHANGED_BEFORE_WREADY	The value of WLAST has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	3.1
AXI3-60115	AXI_WID_CHANGED_BEFORE_WREADY	The value of WID has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	3.1
AXI3-60116	AXI_WLAST_UNKN	WLAST has an X or Z value.	2.3
AXI3-60117	AXI_WREADY_UNKN	WREADY has an X or Z value.	2.3
AXI3-60118	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_12	The WA bit of the cache transaction field should not be HIGH when the Modifiable bit is LOW.	5.1 and 13.8
AXI3-60119	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_13	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1 and 13.8
AXI3-60120	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_4	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1 and 13.8
AXI3-60121	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_5	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1 and 13.8
AXI3-60122	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_8	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1 and 13.8

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60123	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_9	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	5.1 and 13.8
AXI3-60124	AXI_WRITE_BURST_SIZE_VIOLATION	In this write transaction, size has been set greater than the defined data bus width.	4.3
AXI3-60125	AXI_WRITE_DATA_BEFORE_ADDRESS	A write data beat has occurred before the corresponding address phase.	-
AXI3-60126	AXI_WRITE_DATA_UNKN_ON_INVALID_LANE	On a lane whose strobe is 0, WDATA has an X or Z value.	2.3
AXI3-60127	AXI_WRITE_DATA_UNKN_ON_VALID_LANE	On a lane whose strobe is 1, WDATA has an X or Z value.	2.3
AXI3-60128	AXI_RESERVED_AWLOCK_ENCODING	The reserved encoding of 2'b11 should not be used for AWLOCK.	6.1
AXI3-60129	AXI_WRITE_STROBE_ON_INVALID_BYTE_LANES	Write strobe(s) incorrect for the address/size of a fixed transfer.	2.3
AXI3-60130	AXI_WSTRB_CHANGED_BEFORE_WREADY	The value of WSTRB has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	3.1
AXI3-60131	AXI_WSTRB_UNKN	WSTRB has an X or Z value.	2.3
AXI3-60134	AXI_WVALID_DEASSERTED_BEFORE_WREADY	WVALID has been de-asserted before WREADY was asserted.	3.1
AXI3-60135	AXI_WVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A master interface must begin driving WVALID high only at a rising clock edge after ARESETn is HIGH.	11.1.2
AXI3-60136	AXI_WVALID_UNKN	WVALID has an X or Z value.	2.3
AXI3-60137	AXI_ADDR_ACROSS_4K_WITHIN_LOCKED_WRITE_TRANSACTION	Transactions within a locked write sequence should be within the same 4K address boundary.	6.3
AXI3-60138	AXI_ADDR_ACROSS_4K_WITHIN_LOCKED_READ_TRANSACTION	Transactions within a locked read sequence should be within the same 4K address boundary.	6.3

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60139	AXI_AWID_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the awid within the locked transaction.	6.3
AXI3-60140	AXI_ARID_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the arid within the locked transaction.	6.3
AXI3-60141	AXI_AWPROT_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the awprot within the locked transaction.	6.3
AXI3-60142	AXI_ARPROT_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the arprot within the locked transaction.	6.3
AXI3-60143	AXI_AWCACHE_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the awcache within the locked transaction.	6.3
AXI3-60144	AXI_ARCACHE_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the arcache within the locked transaction.	6.3
AXI3-60145	AXI_NUMBER_OF_LOCKED_SEQUENCES_EXCEEDS_2	Number of accesses within a locked sequence should not be more than 2.	6.3
AXI3-60146	AXI_LOCKED_WRITE_BEFORE_COMPLETION_OF_PREVIOUS_WRITE_TRANSACTIONS	A locked write sequence should not commence before completion of all previously issued write addresses.	6.3
AXI3-60147	AXI_LOCKED_WRITE_BEFORE_COMPLETION_OF_PREVIOUS_READ_TRANSACTIONS	A locked write sequence should not commence before completion of all previously issued read addresses.	6.3
AXI3-60148	AXI_LOCKED_READ_BEFORE_COMPLETION_OF_PREVIOUS_WRITE_TRANSACTIONS	A locked read sequence should not commence before completion of all previously issued write addresses.	6.3
AXI3-60149	AXI_LOCKED_READ_BEFORE_COMPLETION_OF_PREVIOUS_READ_TRANSACTIONS	A locked read sequence should not commence before completion of all previously issued read addresses.	6.3
AXI3-60150	AXI_NEW_BURST_BEFORE_COMPLETION_OF_UNLOCK_TRANSACTION	The unlocking transaction should be completed before further any transactions are initiated.	6.3
AXI3-60151	AXI_UNLOCKED_WRITE_WHILE_OUTSTANDING_LOCKED_WRITES	Unlocking write transaction started while outstanding locked write transaction has not completed.	6.3
AXI3-60152	AXI_UNLOCKED_WRITE_WHILE_OUTSTANDING_LOCKED_READS	Unlocking write transaction started while outstanding locked read transaction has not completed.	6.3

Table 5-1. Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60153	AXI_UNLOCKED_READ_WHILE_OUTSTANDING_LOCKED_WRITES	Unlocking read transaction started while outstanding locked write transaction has not completed.	6.3
AXI3-60154	AXI_UNLOCKED_READ_WHILE_OUTSTANDING_LOCKED_READS	Unlocking read transaction started while outstanding locked read transaction has not completed.	6.3
AXI3-60155	AXI_UNLOCKING_TRANSACTION_WITH_AN_EXCLUSIVE_ACCESS	Unlocking transaction cannot be an exclusive access transaction.	6.3
AXI3-60156	AXI_FIRST_DATA_ITEM_OF_TRANSACTION_WRITE_ORDER_VIOLATION	The order in which a slave receives the first data item of each transaction must be the same as the order in which it receives the addresses for the transaction.	8.5
AXI3-60157	AXI_AWLEN_MISMATCHED_ACTUAL_LENGTH_OF_WRITE_DATA_BURST_EXCEEDS_AWLEN	Actual length of data burst has exceeded the burst length specified by AWLEN.	4.2
AXI3-60158	AXI_AWLEN_MISMATCHED_WITH_COMPLETED_WRITE_DATA_BURST	AWLEN value of write address control does not match with corresponding outstanding write data burst length.	4.2
AXI3-60159	AXI_WRITE_LENGTH_MISMATCHED_ACTUAL_LENGTH_OF_WRITE_DATA_BURST_EXCEEDS_AWLEN	The actual length of write data burst exceeds with the length specified by AWLEN.	4.2
AXI3-60160	AXI_WLAST_ASSERTED_DURING_DATA_PHASE_OTHER_THAN_LAST	Wlast must only be asserted during the last data phase.	4.2
AXI3-60161	AXI_WRITE_INTERLEAVE_DEPTH_VIOLATION	Write data bursts should not be interleaved beyond the write interleaving depth.	8.5
AXI3-60162	AXI_WRITE_RESPONSE_WITHOUT_ADDR	Write response should not be sent before the corresponding address has completed.	3.3
AXI3-60163	AXI_WRITE_RESPONSE_WITHOUT_DATA	Write response should not be sent before the corresponding write data burst completed.	3.3
AXI3-60164	AXI_RLAST_VIOLATION	RLAST signal should be asserted along with the final transfer of the read data burst.	4.2

Chapter 6
SystemVerilog Tutorials

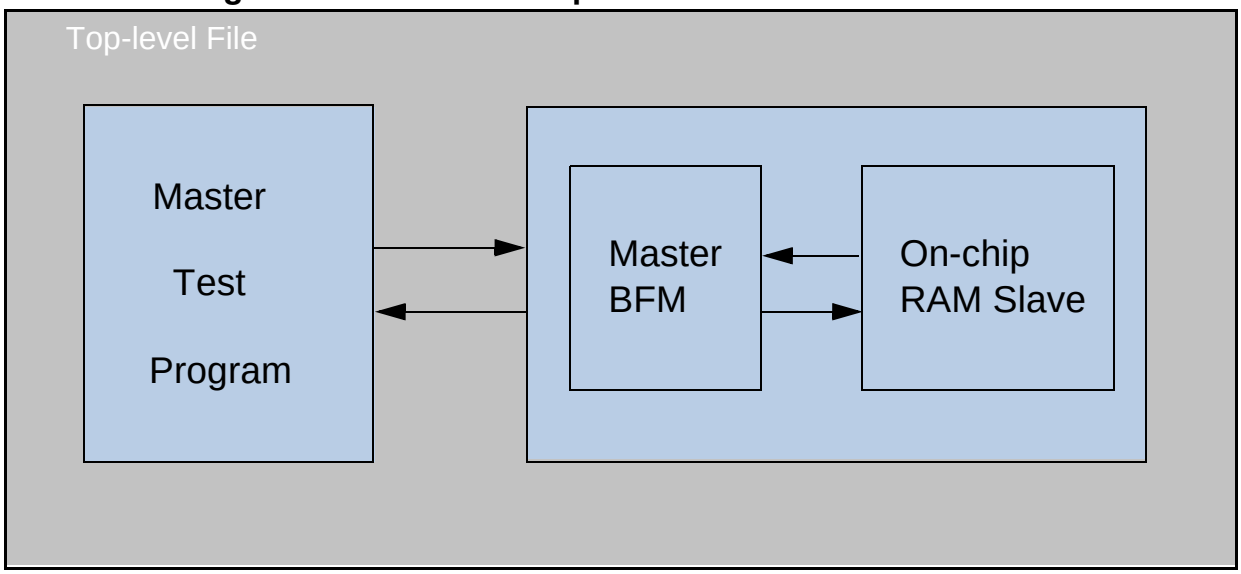
This chapter discusses how to use the Mentor Verification IP Altera Edition master and slave BFMs to verify a master and a slave component. The first example, the slave DUT, is an on-chip RAM that complies with the AXI3 protocol. An AXI3 Master BFM and Master Test Program verify the behavior of the slave DUT. The second example verifies an AXI3 master DUT using an AXI3 Slave BFM and Slave Test Program.

Following this top-level discussion of how you verify a master and a slave component using the Mentor Verification IP Altera Edition is a brief example of how to run Qsys, the powerful system integration tool in Quartus® II software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details on this example refer to "Running Qsys" on page 172.

Verifying a Slave DUT

In this example the DUT is an on-chip RAM slave component connected to a Master BFM at the signal-level. A Master Test Program, written at the transaction-level, generates stimulus via the Master BFM to the slave DUT. Figure 6-1 illustrates a typical top-level testbench environment that verifies the slave component.

Figure 6-1. Slave DUT Top-level Testbench Environment



In this example an AXI3 slave DUT component is connected to an AXI3 Master BFM at the signal-level. The AXI3 Master Test Program configures the AXI3 Master BFM and instructs it to create and execute transactions to the slave DUT. The Master Test Program compares data

written to and read back from the slave DUT and reports any mismatches. The test passes if the read data is correct when compared with the written data.

The top-level file instantiates and connects all the components required to test and monitor the DUT. This file also controls the system clock and reset.

Master Test Program

Using the AXI3 master BFM API, this Master Test Program creates a wide range of stimulus scenarios that test the slave DUT. This tutorial restricts the stimulus to a write transaction followed by a read transaction to the same address to compare the read data with the previously written data. For a complete code example of this Master Test Program, refer to SystemVerilog Master and Slave Test Programs in Appendix A.

The code excerpt in Example 6-1 shows the Master Test Program defining a transaction variable *trans* of type *axi_transaction*, which holds the transaction record. A *timeout* transaction field is configured within the AXI3 Master BFM before waiting for the system reset to be completed. Before executing transactions, an additional system clock cycle is waited on after reset to satisfy the AXI3 protocol requirement specified in Section 11.1.2 of the AMBA AXI Protocol Specification.

Example 6-1. Configuration and Initialization

```
initial
begin
    axi_transaction trans;
    /******************
    ** Configuration **
    ******************/
    begin
      bfm.set_config(AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000);
    end

    /*******************
    ** Initialization **
    *******************/
    bfm.wait_on(AXI_RESET_POSEDGE);
    bfm.wait_on(AXI_CLOCK_POSEDGE);
```

To generate AXI3 protocol traffic, the Master Test Program must create the transaction before executing it. The code excerpt in Example 6-2 uses the AXI3 Master BFM *create_write_transaction ()* function, which provides only the start address argument of the transaction to be created. Because this function call provides only the start address, the burst-length argument automatically defaults to a value of zero—indicating a burst length of a single beat (refer to "AMBA AXI3 Master BFM" on page 14 for more details). The created write transaction is then assigned to the transaction variable *trans*.

This example has an AXI3 data bus width of 32-bits; therefore a single beat of data conveys 4-bytes across the data bus. The *set_data_words()* function sets the '*data_words[0]*' transaction field with the value of 1 on byte lane 1, resulting in a value of 32'h0000_0100. However, the AXI3 protocol permits narrow transfers with the use of the write strobes signal *WSTRB* to indicate which byte lane contains valid write data, and therefore indicates to the slave DUT which data byte lane will be written into memory. Similarly, you can use the *set_write_strobes()* function to set the *write_strobes[0]* transaction field with the value of 4'b0010, indicating that only valid data is being transferred on byte lane 1. The write transaction *trans* then executes on the protocol signals using the *execute_transaction()* function.

All other transaction fields default to legal protocol values (refer to "AXI3 Slave Transaction Fields" on page 35 for more details).

Example 6-2. Write Transaction Creation and Execution

```
/************************
** Traffic generation: **
************************/
// 4 x Writes
// Write data value 1 on byte lanes 1 to address 1.
trans = bfm.create_write_transaction(1);
trans.set_data_words(32'h0000_0100, 0);
trans.set_write_strobes(4'b0010, 0);
$display ( "@ %t, master_test_program: Writing data (1) to address (1)",
$time);

// By default it will run in Blocking mode
bfm.execute_transaction(trans);
```

The code excerpt in Example 6-3 uses the AXI3 Master BFM to read the data that has just been written into the slave memory. The Master Test Program first creates a read transaction using the *create_read_transaction()* function and provides only the start address argument of the transaction to be created. Because this function call provides only the start address, the burst-length automatically defaults to a value of zero—indicating a burst length of a single beat (refer to "AMBA AXI3 Master BFM" on page 14 for more details). The created read transaction is then assigned to the transaction variable *trans*.

The *set_id()* function is then used to set the transaction '*id*' field to be 1 before executing the read transaction *trans* onto the protocol signals.

The read data is obtained using the *get_data_words(0)* function to get the '*data_words[0]*' transaction field value. The result of the read data is compared with the expected data—and a message displays the transcript.

Example 6-3. Read Transaction Creation and Execution

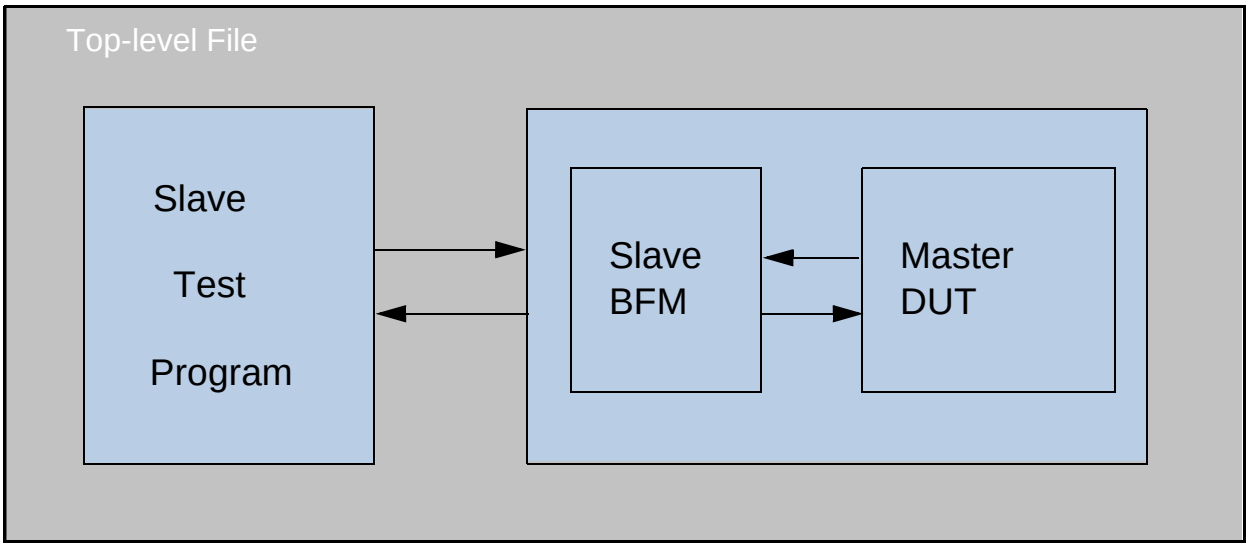
```
// Read data from address 1.
trans = bfm.create_read_transaction(1);
trans.set_id(1);

bfm.execute_transaction(trans);
if (trans.get_data_words(0) == 32'h0000_0100)
    $display ( "@ %t, master_test_program: Read correct data (1) at
        address (1)", $time);
else
    $display ( "@ %t master_test_program: Expected data (1) at address
        1, but got %d", $time, trans.get_data_words(0));
```

Verifying a Master DUT

In this example the DUT is a master component connected to a Slave BFM at the signal-level. A Slave Test Program, written at the transaction-level, generates response stimulus via the Slave BFM to the Master DUT. Figure 6-2 illustrates a typical top-level testbench environment to verify a master component.

Figure 6-2. Master DUT Top-level Testbench Environment



In this tutorial example an AXI3 master DUT component is connected to an AXI3 Slave BFM at the signal-level. The AXI3 Slave Test Program configures the AXI3 Slave BFM and instructs it to create and respond to transactions from the master DUT, storing the write data into an internal memory location.

A top-level file instantiates and connects all the components required to test and monitor the DUT. This program also controls the system clock and reset.

Slave Test Program

The Slave Test Program is a memory model and contains two APIs: a Basic Slave API Definition and an Advanced Slave API Definition.

The Basic Slave API Definition allows you to create a wide range of stimulus scenarios to test a master DUT. This API definition simplifies the creation of slave stimulus based on the AXI3 default response of *OKAY* to master read and write transactions.

The Advanced Slave API Definition allows you to create additional response scenarios to AXI3 transactions. For example, a successful exclusive transaction requires an *EXOKAY* response.

For a complete code listing of the Slave Test Program APIs, refer to Appendix A.

Basic Slave API Definition

The Basic Slave Test Program API contains:

- Functions that read and write a byte of data to internal memory *do_byte_read()* and *do_byte_write()*, respectively.
- Functions to configure the AXI3 protocol channel handshake delays *set_read_address_ready_delay()*, *set_write_address_ready_delay()*, *set_write_data_ready_delay()*, *set_read_data_valid_delay()*, and *set_wr_resp_valid_delay()*.
- Tasks to process read and write transactions, *process_read* and *process_write*, respectively. If you need to create other responses, such as *EXOKAY*, *DECERR*, or *SLVERR*, then you will need to edit these tasks to provide the required response.
- A *slave_mode* transaction field controls the behavior of reading and writing to the internal memory.

The *internal memory* for the slave is defined as a sparse array of 8-bits, so that each byte of data is stored as an address/data pair.

Example 6-4. internal memory

```
// Storage for a memory
bit [7:0] mem [*];
```

The *do_byte_read()* function when called will read a data byte from the internal memory '*mem*' given an address location, as demonstrated in Example 6-5.

You may edit this function to modify the way the read data is extracted from the internal memory.

Example 6-5. do_byte_read()

```
// Function : do_byte_read
// Function to provide read data byte from memory at
// particular input address
function bit[7:0] do_byte_read(addr_t addr);
   return mem[addr];
endfunction
```

The *do_byte_write()* function when called writes a data byte to the internal memory '*mem*' given an address location, as Example 6-6 illustrates.

You may edit this function to modify the way the write data is stored in the internal memory.

Example 6-6. do_byte_write()

```
// Function : do_byte_write
// Function to write data byte to memory at particular
// input address
function void do_byte_write(addr_t addr, bit [7:0] data);
   mem[addr] = data;
endfunction
```

The *set_read_address_ready_delay()* function when called configures the ARREADY handshake signal to be delayed by a number of ACLK cycles which extends the length of the read address phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "Delay Mode" on page 10 for details). Example 6-7 demonstrates setting the *ARREADY* signal delay by 4 *ACLK* cycles.

You may edit this function to change the *ARREADY* signal delay.

Example 6-7. set_read_address_ready_delay()

```
// Function : set_read_address_ready_delay
// This is used to set read address phase ready delay
// to extend phase
function void set_read_address_ready_delay(axi_transaction trans);
   trans.set_address_ready_delay(4);
endfunction
```

The *set_write_address_ready_delay()* function when called configures the *AWREADY* handshake signal to be delayed by a number of *ACLK* cycles, which extends the length of the write address phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "Delay Mode" on page 10 for details). Example 6-8 demonstrates setting the *AWREADY* signal delay by 2 *ACLK* cycles.

You may edit this function to change the *AWREADY* signal delay.

Example 6-8. set_write_address_ready_delay()

```
// Function : set_write_address_ready_delay
// This is used to set write address phase ready delay
// to extend phase
function void set_write_address_ready_delay(axi_transaction trans);
  trans.set_address_ready_delay(2);
endfunction
```

The *set_write_data_ready_delay()* function when called configures the *WREADY* signal handshake to be delayed by a number of *ACLK* cycles which extends the length of each write data phase (beat) within a write data burst. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "Delay Mode" on page 10 for details).

For each write data phase (beat) the delay value of the *WREADY* signal is stored in an element of the *data_ready_delay[]* array for the transaction, as demonstrated in Example 6-9.

You may edit this function to change the *WREADY* signal delays.

Example 6-9. set_write_data_ready_delay()

```
// Function : set_write_data_ready_delay
// This will set the ready delays for each write data phase
// in a write data burst
function void set_write_data_ready_delay(axi_transaction trans);
   for (int i = 0; i < trans.data_ready_delay.size(); i++)
      trans.set_data_ready_delay(i, i);
endfunction
```

The *set_read_data_valid_delay()* function when called configures the *RVALID* signal to be delayed by a number of *ACLK* cycles with the effect of delaying the start of each read data phase (beat) within a read data burst. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "Delay Mode" on page 10 for details).

For each read data phase (beat) the delay value of the *RVALID* signal is stored in an element of the *data_valid_delay[]* array for the transaction, as demonstrated in Example 6-10.

You may edit this function to change the *RVALID* signals delays.

Example 6-10. set_read_data_valid_delay()

```
// Function : set_read_data_valid_delay
// This is used to set read response phase valid delays to start
// driving read data/response phases after specified delay.
function void set_read_data_valid_delay(axi_transaction trans);
  for (int i = 0; i < trans.data_valid_delay.size(); i++)
    trans.set_data_valid_delay(i, i);
endfunction
```

The *set_wr_resp_valid_delay()* function when called configures the *BREADY* signal handshake to be delayed by a number of *ACLK* cycles which extends the length of the write response phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "Delay Mode" on page 10 for details). Example 6-11 below demonstrates setting the *BREADY* signal delay by 2 *ACLK* cycles.

You may edit this function to change the *BREADY* signals delay.

Example 6-11. set_wr_resp_valid_delay()

```
// Function : set_wr_resp_valid_delay
// This is used to set write response phase valid delay to start
// driving write response phase after specified delay.
function void set_wr_resp_valid_delay(axi_transaction trans);
   trans.set_write_response_valid_delay(2);
endfunction
```

There is a *slave_mode* transaction field that you can configure to control the behavior of reading and writing to the internal memory. It has two modes *AXI_TRANSACTION_SLAVE* and *AXI_PHASE_SLAVE*.

Example 6-12. slave_mode

```
// Enum type for slave mode
// AXI_TRANSACTION_SLAVE - Works at burst level (write data is received at
//                         burst and read data/response is sent in burst)
// AXI_PHASE_SLAVE       - Write data and read data/response is serviced
//                         at phase level
   typedef enum bit
{
      AXI_TRANSACTION_SLAVE = 1'b0,
      AXI_PHASE_SLAVE       = 1'b1
} axi_slave_mode_e;


// Slave mode selection : Default is transaction-level slave
axi_slave_mode_e slave_mode = AXI_TRANSACTION_SLAVE;
```

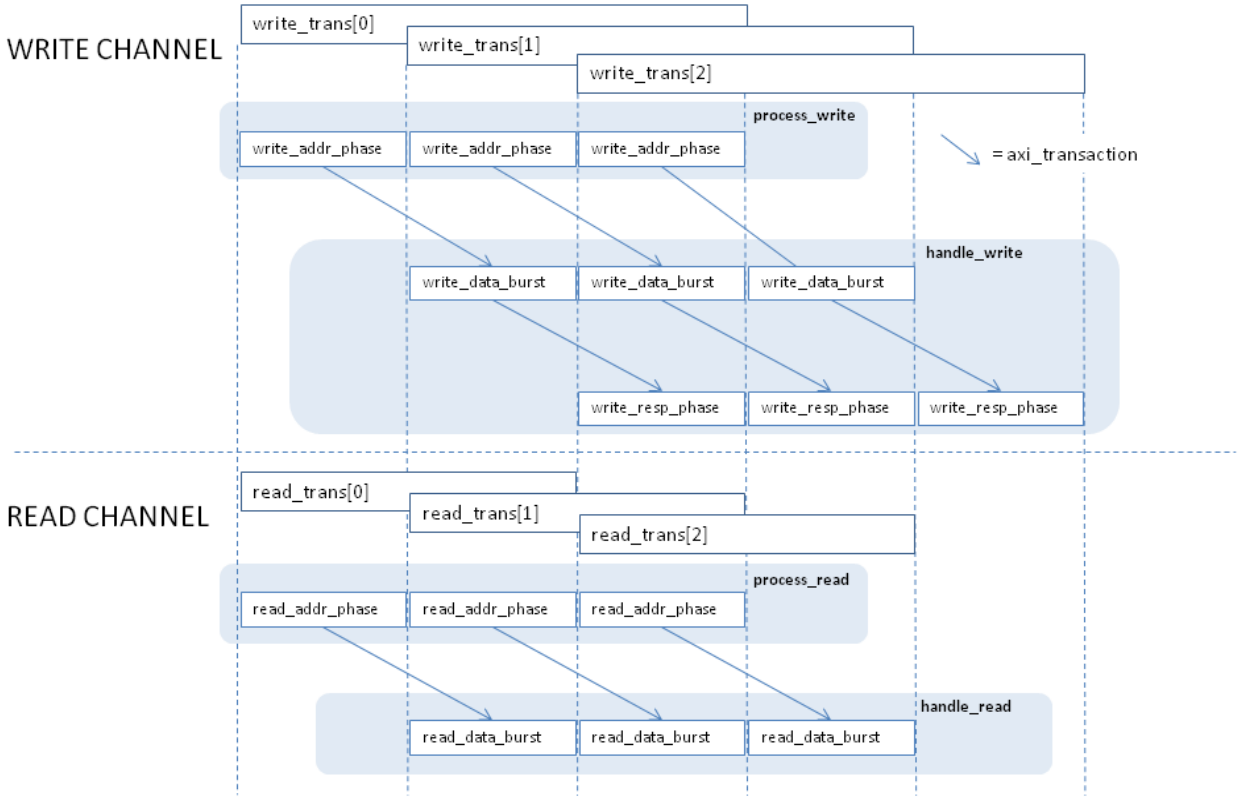
The default *AXI_TRANSACTION_SLAVE* mode "saves up" an entire data burst and modifies the Slave Test Program internal memory in zero time for the whole burst. Therefore, a read from internal memory is buffered at the beginning of the read burst for the whole burst. The buffered read data is then transmitted over the protocol signals to the master on a phase-by-phase (beat-by-beat) basis. For a write, the write data burst is buffered on a phase-by-phase (beat-by-beat) basis for the whole burst. Only at the end of the write burst are the buffered contents written to the internal memory.

The *AXI_PHASE_SLAVE* mode changes the Slave Test Program internal memory on each data phase (beat). Therefore, a read from the internal memory occurs only when the read data phase (beat) actually starts on the protocol signals. For a write, data is written to the internal memory as soon as each individual write data phase (beat) completes.

Note

In addition to the above functions, you can configure other aspects of the AXI3 Slave BFM by using the functions: *"set_config()"* on page 37 and *"get_config()"* on page 38.

Figure 6-3. API Pipelining



Using the Basic Slave Test Program API

As described in the Basic Slave API Definition section, there are a set of tasks and functions that you can use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing. However, consider the following configurations when using the Slave Test program.

- *slave_mode* - The read and write channel interaction may cause simultaneous read and write transactions to occur at the same address. With the default *slave_mode* setting the read transaction data burst is buffered at the start of the burst and the write data burst is buffered at the end of the burst. This may result in the read data being stale at the time it is transmitted over the protocol signals. If this is an undesirable feature, then set the *Slave_mode* to be *AXI_PHASE_SLAVE*.

- *delay_mode* - By default the handshake **READY* signal will always follow, or be simultaneous with, the **VALID* signal. By configuring the *delay_mode* to be *AXI_TRANS2READY* **READY* before **VALID* scenarios can be achieved.

Advanced Slave API Definition

The remaining section of this tutorial presents a walk-through of the Advanced Slave API. It consists of two main procedures, *process_read* and *process_write*, within the Slave Test Program.

Note

You are not required to edit the following Advance Slave API unless you require a different response than the default (*OKAY*) response.

Within an initial block, the Slave Test Program waits for the *ARESETn* signal to be deactivated before processing any read, or write, transactions simultaneously within a fork-join block, as demonstrated in Example 6-13.

Example 6-13. Initialization and Transaction Processing

```
initial
begin
  // Initialisation
  wait (bfm.ARESETn == 1);

  // Traffic generation
  fork
    process_read;
    process_write;
  join
end
```

The process_read task loops forever, processing read transactions as they occur from the master. It defines a local transaction variable '*read_trans*' of type '*axi_transaction*' to store a record of the read transaction while it is being processed. It then uses the Slave BFM function *create_slave_transaction()* to create a read transaction and assign it to the local '*read_trans*' record.

The *set_read_address_ready_delay()* function is called to configure the delay for the *ARREADY* signal before getting the read address phase using the Slave BFM *get_read_addr_phase()* task.

The subsequent *fork-join_none* block performs a nonblocking statement so that the *process_read* task can begin again to create another read transaction and get another read address phase before the current read transaction has completed. This permits concurrent read transactions to occur if the master issues a series of read address phases before any previous read transactions have completed.

Within the *fork-join_none* block the '*read_trans'* record is passed into the *handle_read()* function via the variable '*t*'.

Example 6-14. process_read

```
// Task : process_read
// This method keep receiving read address phase and calls another
// method to process received transaction.
task process_read;
  forever
  begin
    axi_transaction read_trans;

    read_trans = bfm.create_slave_transaction();
    set_read_address_ready_delay(read_trans);
    bfm.get_read_addr_phase(read_trans);

    fork
      begin
        automatic axi_transaction t = read_trans;
        handle_read(t);
      end
    join_none
    #0;
  end
endtask
```

The *set_read_data_ready()* function calls the *set_data_valid_delay()* function in the Slave BFM. It configures the delay for the assertion of *ARVALID* signal for each read data phase (beat) of a read burst.

Example 6-15. set_read_data_ready()

```
// Function : set_read_data_valid_delay
// This is used to set read response phase valid delays to start
// driving read data/response phases after specified delay.
function void set_read_data_valid_delay(axi_transaction trans);
  for (int i = 0; i < trans.data_valid_delay.size(); i++)
    trans.set_data_valid_delay(i, i);
endfunction
```

The *handle_read* task gets the data from the internal memory in either bursts or phases depending on the *slave_mode* configuration. It's '*read_trans*' argument contains the record of the read transaction up to the point of this task call, namely the content of the read address phase.

The call to *set_read_data_valid_delay()* configures the assertion of the *ARVALID* signal delay.

The task then loops for the number of addresses defined by calling the *get_read_addr()* helper function in the Slave BFM, assigning the local '*mem_data*' variable with read data by calling the *do_byte_read()* function.

The Slave BFM helper function *set_read_data()* then fills a complete read data bus width of data into the Slave BFM *data_words[]* array. At this point a read data phase is executed on to the read data channel if the *slave_mode* setting is *AXI_PHASE_SLAVE*, otherwise it executes a complete read data burst only when the Slave BFM *data_words[]* array contains a complete burst of read data.

Example 6-16. handle_read

```
// Task : handle_read
// This method reads data from memory and send read data/response
// either at burst or phase level depending upon slave working
// mode.
task automatic handle_read(input axi_transaction read_trans);
  addr_t addr[];
  bit [7:0] mem_data[];

  set_read_data_valid_delay(read_trans);

  for(int i = 0; bfm.get_read_addr(read_trans, i, addr); i++)
  begin
    mem_data = new[addr.size()];
    for (int j = 0; j < addr.size(); j++)
      mem_data[j] = do_byte_read(addr[j]);

    bfm.set_read_data(read_trans, i, addr, mem_data);

    if (slave_mode == AXI_PHASE_SLAVE)
      bfm.execute_read_data_phase(read_trans, i);
  end
    if (slave_mode == AXI_TRANSACTION_SLAVE)
      bfm.execute_read_data_burst(read_trans);
endtask
```

The processing of write transactions in the Slave Test Program works in a similar way as that described for read transactions. Processing a write transaction requires both a *process_write* task and a *handle_write* task.

The main difference is that the write transaction handling gets the write data burst and stores it in the Slave Test Program *internal memory* depending on the *slave_mode* setting, adhering to the state of the write strobes signal *WSTRB*. There is also an additional write response phase that is required for the AXI3 write channel.

Example 6-17. process_write

```
// Task : process_write
// This method keep receiving write address phase and calls another
// method to process received transaction.
task process_write;
  forever
  begin
    axi_transaction write_trans;

    write_trans = bfm.create_slave_transaction();
    set_write_address_ready_delay(write_trans);
    bfm.get_write_addr_phase(write_trans);

    fork
      begin
        automatic axi_transaction t = write_trans;
        handle_write(t);
      end
    join_none
    #0;
  end
endtask
```

Example 6-18. handle_write

```
// Task : handle_write
// This method receive write data burst or phases for write
// transaction depending upon slave working mode, write data to
// memory and then send response
task automatic handle_write(input axi_transaction write_trans);
  addr_t addr[];
  bit [7:0] data[];
  bit last;

  set_write_data_ready_delay(write_trans);

  if (slave_mode == AXI_TRANSACTION_SLAVE)
        begin
    bfm.get_write_data_burst(write_trans);

    for( int i = 0; bfm.get_write_addr_data(write_trans,
         i, addr, data); i++ )
    begin
      for (int j = 0; j < addr.size(); j++)
         do_byte_write(addr[j], data[j]);
    end
  end
  else
  begin
    for(int i = 0; (last == 1'b0); i++)
    begin
      bfm.get_write_data_phase(write_trans, i, last);
```

```
        void'(bfm.get_write_addr_data(write_trans, i, addr, data));
        for (int j = 0; j < addr.size(); j++)
          do_byte_write(addr[j], data[j]);
      end
    end

    set_wr_resp_valid_delay(write_trans);
    bfm.execute_write_response_phase(write_trans);
  endtask
```

Chapter 7
VHDL API Overview

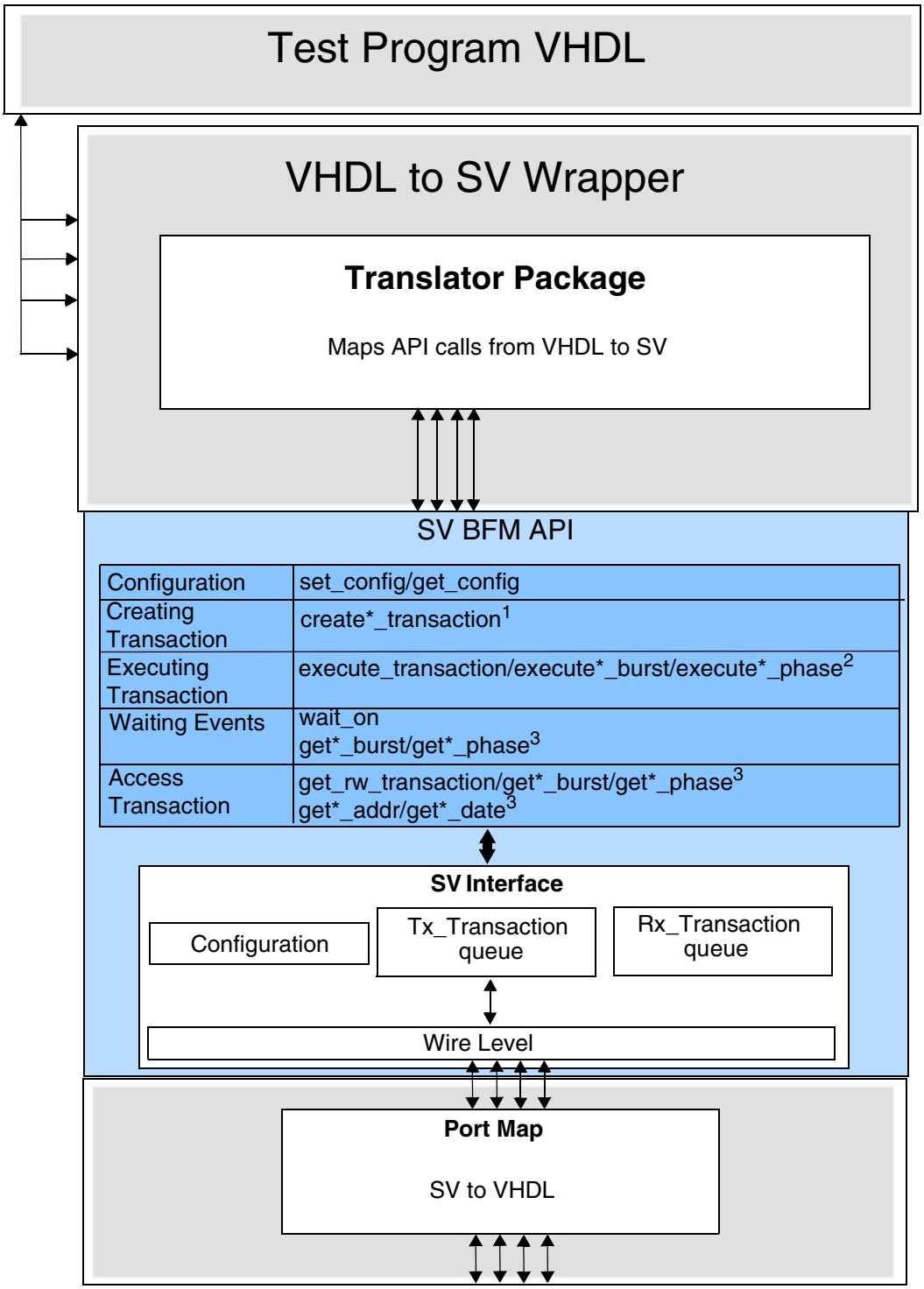
This section describes how the VHDL Application Programming Interface (API) functions for all the BFM (master, slave, and monitor) components. For each BFM, you can configure protocol transaction fields that execute on the protocol signals and control the operational transaction fields that permit delays between the handshake signals for each of the five address, data, and response channels.

The BFMs also let you set wait intervals for events that occur on the system clock and reset signals, plus let you get and/or set information about a particular transaction.

Note

The VHDL API is built on the SystemVerilog API. An internal VHDL to SystemVerilog (SV) wrapper casts the VHDL BFM API procedure calls to the SystemVerilog BFM API tasks and functions.

Figure 7-1. VHDL BFM Internal Structure



Notes: 1. Refer to the create*_transaction()
2. Refer to the execute_transaction(), execute*_burst(), execute*_phase()
3. Refer to the get*()

Configuration

To configure a BFM, you set timing, configuration, error reporting, and other attributes of the BFM.

set_config()

Each master, slave, and monitor BFM has a *set_config()* procedure, which sets the configuration of the BFM. Refer to the BFM *set_config()* procedures that are relevant to the master, slave, or monitor BFM to ensure you specify valid configurations.

For example, the following code sets the burst timeout factor for a transaction:

```
-- Setting the burst timeout factor to 1000
Set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi_tr_if_0(bfm_index))
```

In the above example, the *bfm_index* specifies the BFM.

get_config

Each master, slave, and monitor BFM has a *get_config()* procedure, which returns the configuration of the BFM. Refer to the specific BFM *get_config()* procedure for details.

For example, to get the burst timeout factor value for a transaction:

```
-- Getting the burst timeout factor
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
      axi_tr_if_0(bfm_index))
```

In the above example, the *bfm_index* specifies the BFM.

Creating Transactions

An AXI3 transaction consists of many protocol transaction fields like address, length, data, size, and strobes, etc. However, the master API allows you to create a master transaction by providing only the address and burst length protocol transaction fields as arguments for a read or write transaction. All other protocol transaction fields automatically default to valid AXI3 protocol values to create a complete transaction. If you change a master protocol transaction field value from its default, the changed value is then valid for all future transactions until a new value is set. Refer to "create_write_transaction()" on page 129 for default protocol transaction field values.

The slave API also allows you to create a slave transaction by providing no protocol transaction fields as arguments. All protocol transaction fields automatically default to valid AXI3 protocol values to create a complete slave transaction. If you change a slave protocol transaction field value from its default, the changed value is then valid for all future transactions until a new

value is set. Refer to "create_slave_transaction()" on page 150 for default protocol transaction fields values.

create*_transaction()

There are two master API procedures available to create AXI3 transactions, *create_write_transaction()* and *create_read_transaction()*, a single slave API procedure *create_slave_transaction()* and a single monitor API procedure *create_monitor_transaction()*.

For example, the following code shows you how to create a simple write transaction with a start address of 1 and a single beat data burst with a data value of 2:

```
-- Define local variables to hold the transaction ID
-- and data word.
variable tr_id: integer;
variable data_words :  std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);

-- Create a master write transaction and set data_word value
create_write_transaction(1, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"00000200";
set_data_words(data_words, tr_id, index, axi_tr_if_0(index));
```

For example, the following code shows you how to create a simple slave transaction:

```
-- Define a local variable write_trans to hold the transaction ID
variable write_trans : integer;

-- Create a slave transaction
create_slave_transaction(write_trans, index, axi_tr_if_0(index));
```

Executing Transactions

Executing transaction tasks push the created transactions into the internal transaction queues, shown in Figure 7-1, for execution onto the protocol signals.

execute_transaction(), execute*_burst(), execute*_phase()

Typically, you use the *execute_transaction()* procedure at the master BFM and *execute*_burst()* and *execute*_phase()* tasks at the slave BFM.

For example, the following code shows you how to execute a master write transaction:

```
-- By default the execution of a transaction will block
execute_transaction(tr_id, index, axi_tr_if_0(index));
```

For example, the following code shows you how to execute a slave write response phase:

```
-- By default the execution of a phase will block
execute_write_response_phase(write_trans, index, axi_tr_if_2(index));
```

Waiting Events

Each master, slave, and monitor BFM has a *wait_on()* task that permits the user to wait for an AXI3 *ACLK* or *ARESETn* event to occur before proceeding.

wait_on()

For example, the following code specifies that the master has *wait_on()* at the positive edge of the *ARESETn* signal:

```
-- Block test program execution until the positive edge of the clock
wait_on(AXI_RESET_POSEDGE, index, axi_tr_if_0(index));
```

Access Transaction

Each master, slave, and monitor BFM can access a complete or partially complete transaction by using various *get*()* tasks to gather information.

get*()

For example, a slave may use a received write address phase to form the response of the transactions shown below.

```
-- Wait for a write address phase;
get_write_addr_phase(write_trans, index, axi_tr_if_0(index));

...

execute_write_response_phase(write_trans, index, axi_tr_if_2(index));
```

Operational Transaction Fields

Operational transaction fields control the way in which a transaction is executed onto the protocol signals, with the exception of the *gen_write_strobes* transaction field. Transaction fields also indicate when a data phase (beat) or a complete transaction has completed.

Automatic Correction of Byte Lane Strobes

The AXI3 protocol permits unaligned and narrow write transfers by using byte lane strobe (*WSTRB)* signals to indicate which byte lanes contain valid data per data phase (beat).

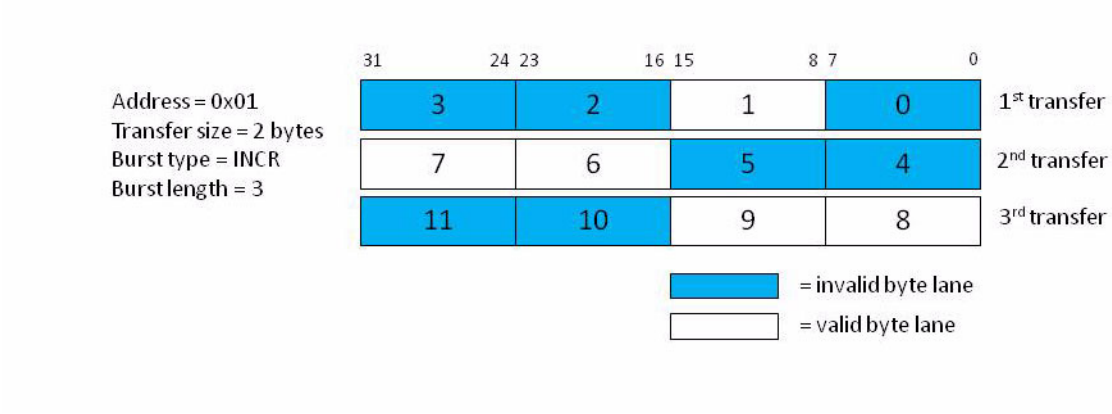
When you create a write transaction in your test program, the array variable *write_strobes[]* is available to store the write strobe values for each write data phase (beat) within the transaction. To assist you in creating the correct byte lane strobes per write data phase (beat), automatic correction of previously set *write_strobes[]* array elements is performed by default during the execution of a write transaction or write data burst. You can disable this default behavior by

setting the transaction field *gen_write_strobes = 0*. This setting allows any previously set *write_strobes[]* array elements to pass through uncorrected onto the protocol *WSTRB* signals. In this mode, which does not auto-correct the *write_stobes []* array elements, you are responsible for setting the correct *write_strobes[]* array elements for the whole transaction.

The automatic correction algorithm performs a bit-wise AND operation on any previously set *write_strobes[]* array elements. When making corrections, the automatic correction algorithm uses the equations described in the AMBA AXI Protocol Specification, version 2.0, section 4.5, that define valid write data byte lanes for legal protocol. Therefore, if you require automatic generation of all *write_strobes[]* array elements for a write transaction you must previously set all *write_strobes[]* array elements to be 1, indicating that all bytes lanes initially contain valid write data, prior to execution of the write transaction. Automatic correction will then set the relevant *write_strobes[]* array elements to '0' to produce legal protocol *WSTRB* signals.

For example, Figure XXX below demonstrates byte lanes that may contain valid data for a write transaction that has a start address = 0x01, size = 0b001 (2 bytes), type = INCR and length = 0b0010 (3 beats), for a 32-bit write data bus.

Figure 7-2. Valid Data on Byte Lanes During a Write Transaction



In the above example, if you set all *write_strobes[]* array elements to 1 prior to executing the write transaction, automatic correction produces the following results during execution of the transaction.

	Prior to execution		During execution
1st data phase -	write_strobes[0]=0b1111	->	write_strobes[0]=0b0010
2nd data phase -	write_strobes[1]=0b1111	->	write_strobes[1]=0b1100
3rd data phase -	write_strobes[2]=0b1111	->	write_strobes[2]=0b0011

If you randomly set all *write_strobes[]* array elements to 0 or 1, prior to executing the write transaction, automatic correction **only** corrects those *write_strobes[]* array elements that were previously set to 1, as shown below.

	Prior to execution		During execution
1st data phase -	write_strobes[0]=0b1010	->	write_strobes[0]=0b0010
2nd data phase -	write_strobes[1]=0b1010	->	write_strobes[1]=0b1000
3rd data phase -	write_strobes[2]=0b1010	->	write_strobes[2]=0b0010

Note

To automatically generate all *WSTRB* signals for a write transaction, set all *write_strobes[]* array elements to 1 prior to execution of the write transaction or write data burst.

Operation Mode

By default, each read or write transaction will perform a blocking operation which prevents a subsequent transaction from starting until the present active transaction completes.

You can configure this behavior to be nonblocking by setting the *operation_mode* transaction field to the enumerate type value *AXI_TRANSACTION_NON_BLOCKING* instead of the default *AXI_TRANSACTION_BLOCKING*. Typically, you set this configuration by calling the "create_write_transaction()" on page 129 and "create_read_transaction()" on page 132 tasks at the master BFM and the "create_slave_transaction()" on page 150 at the slave BFM.

Handshake Delay

You can configure the delay between the **VALID* and **READY* handshake signals for each of the five address, data, and response channels. These delays can be set on a per phase cycle basis for a particular transaction, measured from the positive edge of *ACLK* when **VALID* is asserted within a transaction, or the completion of a previous transaction phase (**VALID* and **READY* both asserted).

Handshake Signal Delay Transaction Fields

There are many transaction fields that you can configure to produce the desired handshake delay pattern for a particular transaction phase on any of the five protocol channels. Typically, handshaking is set such that the master BFM configures the **VALID* and **READY* signal delays that the master asserts, and the slave BFM configures the **VALID* and **READY* signal delays that it asserts. Table 7-1 specifies which operational delay transaction fields are configured by the master and slave BFMs.

Table 7-1. Handshake Signal Delay Transaction Fields

Signal	Operational Transaction Field	Configuration BFM
AWVALID	address_valid_delay	Master
AWREADY	address_ready_delay	Slave
WVALID	data_valid_delay	Master
WREADY	data_ready_delay	Slave
BVALID	write_response_valid_delay	Slave
BREADY	write_response_ready_delay	Master
ARVALID	address_valid_delay	Master
ARREADY	address_ready_delay	Slave
RVALID	data_valid_delay	Slave
RREADY	data_ready_delay	Master

Note

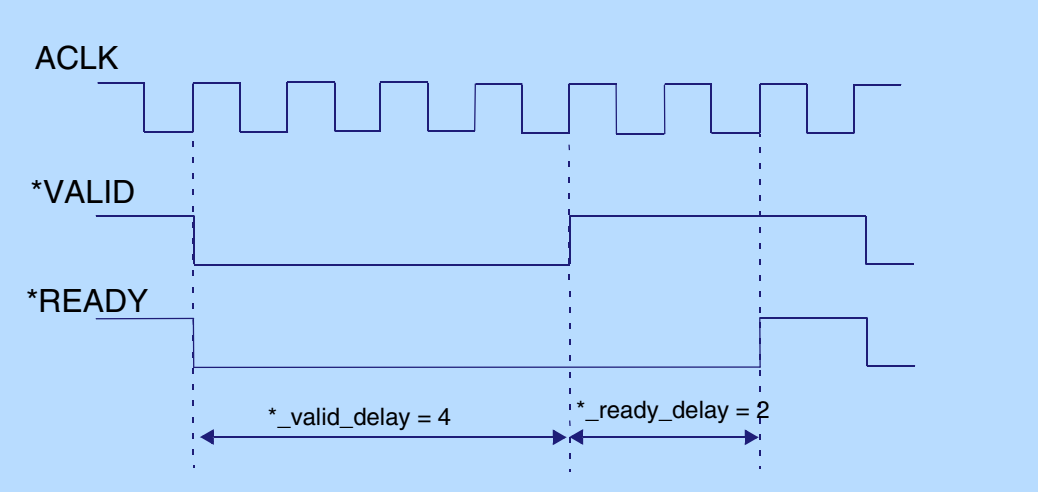
The data channel handshake signal transaction fields (*data_valid_delay[]* and *data_ready_delay[]*) are defined as arrays so that the **VALID* to **READY* delay can be configured on a per data phase (beat) basis within a transaction.

Delay Mode

The delay mode can be configured on a per transaction basis via the *delay_mode* operational transaction field. This transaction field can be configured to the enumerated type values of *AXI_VALID2READY* (default), or *AXI_TRANS2READY*.

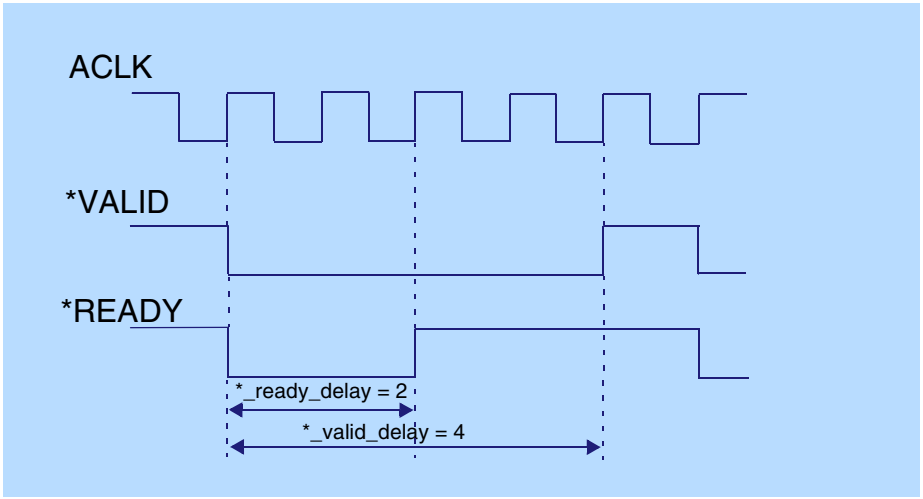
The default configuration (*delay_mode* = *AXI_VALID2READY*) corresponds to the delay measured from the positive edge of *ACLK* when **VALID* is asserted within a transaction. Figure 7-3 demonstrates how to achieve a **VALID* asserted before **READY* handshake.

Figure 7-3. Operational Transaction Field delay_mode = AXI_VALID2READY



The other configuration (*delay_mode* = *AXI_TRANS2READY*) corresponds to the delay measured from the completion of a previous transaction phase (**VALID* and **READY* both asserted). Figure 7-4 demonstrates how to achieve a **READY* before **VALID* handshake.

Figure 7-4. Operational Transaction Field delay_mode = AXI_TRANS2READY



Data Beat Done

There is a *data_beat_done* transaction field in each transaction, defined as an array, to indicate when each data phase (beat) has completed. Each element of the *data_beat_done* array is set to 1 when each data phase (beat) has completed within a data burst.

You can call the *get_read_data_phase()* procedure at the master BFM to determine how many beats of a read data burst have completed by analyzing how many elements of the *data_beat_done* array have been set to 1. Similarly, the *get_write_data_phase()* task can be called at the slave BFM to analyze a write data burst.

Transaction Done

There is a *transaction_done* transaction field in each transaction which indicates when the transaction has completed.

You can call the *get_read_data_burst()* task at the master BFM to determine whether a transaction has completed for a read transfer, and *get_write_response_phase()*. You can also do this at the master BFM, to determine whether a transaction has completed for a write transfer.

Chapter 8
AMBA AXI VHDL BFMs

This section provides information about the AMBA VHDL BFMs for the Mentor Verification IP Altera Edition.

Overloaded Procedure Common Arguments

The AMBA AXI VHDL BFMs use VHDL procedure overloading and therefore have a number of prototype definitions per procedure. Their arguments are unique to each procedure and concern the protocol or operational transaction fields for a transaction. Also arguments that are common to all procedures may be optional. Descriptions of these common procedure follow:

- *transaction_id* - is an index number that identifies a specific transaction. Each new transaction automatically increments the index number until the maximum value of 255 is reached, at which point the index number automatically wraps to zero. The *transaction_id* identifies unique transactions when there are a number of concurrently active transactions.
- *bfm_id* - is a unique identification number for each master, slave, and monitor BFM within a multiple BFM test environment.
- *path_id* (optional) - is a unique identifier for each parallel process within a multiple process testbench. It must be used to permit any testbench stimulus to replicate the pipelining features of a protocol within a VHDL testbench. If there is no pipelining performed in the testbench stimulus (a single process) then you are not required to provide the *path_id* argument to the procedure.
- *tr_if* - is a signal definition that passes the content of a transaction between the VHDL and SystemVerilog environments.

AMBA AXI3 Master BFM

The AMBA AXI3 Master BFM implements the full AMBAAXI3 protocol. In addition to the standard protocol, it supports user sideband signals *AWUSER* and *ARUSER*.

AXI3 Master Timing

For detailed timing diagrams of the protocol bus activity refer to the AMBA AXI Protocol Specification to reference the following details of the BFM and events.

The specification does not define any timescale, or clock period, with signal events sampled and driven at rising *ACLK* edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

AXI3 Master Configuration

The AMBA AXI3 Master BFM supports the full range of signals defined for the AMBA AXI3 protocol. You can customize the AMBA AXI3 Master interface using the configuration fields described in Table 8-1.

Table 8-1. AXI3 Master BFM Configuration

Configuration Field	Description
Signal Width #Parameter	
AXI_ADDRESS_WIDTH	Address width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
AXI_RDATA_WIDTH	Read data width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 1024.
AXI_WDATA_WIDTH	Write data width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 1024.
AXI_ID_WIDTH	ID width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
Timing Variables	
AXI_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.
AXI_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.
AXI_CONFIG_MAX_ TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_ DATA_TRANSFER	The maximum number of write data beats that the AXI interface can generate as part of write data burst of write transfer. Default: 1024.
AXI_CONFIG_BURST_TIMEOUT_ FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
Master Attributes	
AXI_CONFIG_WRITE_CTRL_TO_ DATA_MINTIME	The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_WRITE_DATA_ TO_CTRL_MINTIME	The minimum delay between the start of a write data phase to the start of a write response phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_DEFAULT_ UNDER_RESET	The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true

Table 8-1. AXI3 Master BFM Configuration

Slave Attributes	
AXI_CONFIG_SUPPORT_ EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_ UNDER_RESET	The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
Error Detection	
AXI_CONFIG_ENABLE_ERRORS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to AXI3 Master Timing for details of simulator time-steps.

AXI3 Master Assertions

The master BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

By default all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the set_config() command as the following example illustrates.

set_config(AXI_CONFIG_ENABLE_ERRORS,0,bfm_index, axi_tr_if_0(bfm_index));

Alternatively, you can disable individual built-in assertions by using a sequence of get_config() and set_config() commands on the respective assertion. For example, to disable assertion checking for the *AWLOCK* signal changing between the *AWVALID* and *AWREADY* handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI_CONFIG_ENABLE_ERROR,config_assert_bitvector, bfm_index,
axi_tr_if_0(bfm_index));

-- Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] := 0;
```

```
-- Set the new value of the assertion bit vector
set_config(AXI_CONFIG_ENABLE_ERROR,config_assert_bitvector, bfm_index,
axi_tr_if_0(bfm_index));
```

Note

Do not confuse the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector with the *AXI_CONFIG_ENABLE_ERRORS* global enable/disable.

To re-enable the *AXI_LOCK_CHANGED_BEFORE_AWREADY* assertion, following the above code sequence, assign the assertion within the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector to 1.

For a complete listing of assertions, refer to "AXI3 Assertions" on page 87.

VHDL AXI3 Master Application Program Interface

This section describes the VHDL AMBA AXI3 Master API for the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the AXI3 Master BFM.

Table 8-2. VHDL AXI3 Master set_config() Procedure

Prototype

```
procedure set_config
(
   config_name    : in std_logic_vector(7 downto 0);
   config_val     : in std_logic_vector(AXI_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id         : in integer;
   path_id        : in axi_path_t; --optional
   signal tr_if   : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

config_name

AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "AXI3 Master Configuration" on page 124 for description and valid values.

config_value

bfm_id path_id (optional) tr_if	For descriptions, refer to "Overloaded Procedure Common Arguments" on page 123

Returns

None

Usage

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi_tr_if_0(bfm_index));
Set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi_tr_if_0(bfm_index));
```

get_config()

This procedure gets the configuration of the AXI3 Master BFM.

Table 8-3. VHDL AXI3 Master get_config() Procedure

Prototype

```
procedure get_config
(
      config_name    : in std_logic_vector(7 downto 0);
      config_val     : in
      std_logic_vector(AXI_MAX_BIT_SIZE-1 downto
      0)|integer;
      bfm_id         : in integer;
      path_id        : in axi_path_t; --optional
      signal tr_if   : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

config_name

AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "AXI3 Master Configuration" on page 124 for description and valid values.

config_value

bfm_id path_id (optional) tr_if	Refer to "AXI3 Master Configuration" on page 124 for description and valid values.

Returns

config_val

Usage

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
```

create_write_transaction()

This procedure creates a write transaction with a user supplied start address and optional burst length. All other transaction fields default to legal protocol values, unless assigned values.

Table 8-4. VHDL AXI3 Master create_write_transaction Procedure

Prototype

```
procedure create_write_transaction
(
      addr             : in
      std_logic_vector(AXI_MAX_BIT_SIZE-1 downto
      0)|integer;
      burst_length     : in integer; --optional
      transaction_id   : out integer;
      bfm_id           : in integer;
      path_id          : in axi_path_t; --optional
      signal tr_if     : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments	addr	Start address
	burst_length	Burst length (default = 0)
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"
Protocol Transaction Fields	size	Burst size (defaults to width of bus) – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
	burst	Burst type – AXI_FIXED; AXI_INCR (default); AXI_WRAP; AXI_BURST_RSVD;
	lock	Burst lock – AXI_NORMAL (default); AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;

Table 8-4. VHDL AXI3 Master create_write_transaction Procedure (cont.)

Protocol Transaction Fields (continued)	cache	Burst cache – AXI_NONCACHE_NONBUF (default); AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY ; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;
	prot	Burst protection – AXI_NORM_SEC_DATA (default); AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
	id	Burst ID
	data_words[ ]	Data words array
	write_strobes[ ]	Write strobes array – Each element 0 or 1
	resp	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;

Table 8-4. VHDL AXI3 Master create_write_transaction Procedure (cont.)

Operational Transaction Fields	gen_write_strobes	Generate write strobes flag – 0 = user supplied write strobes 1 = auto-generated write strobes (default)
	operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING (default);
	delay_mode	AXI_VALID2READY (default); AXI_TRANS2READY;
	write_data_mode	AXI_DATA_AFTER_ADDRESS (default); AXI_DATA_WITH_ADDRESS;
	address_valid_delay	Write address channel AWVALID delay in ACLK cycles for this transaction (default = 0)
	data_valid_delay[]	Write data channel WVALID delay array in ACLK cycles for this transaction (default = 0 for all elements)
	write_response_ready_delay	Write response channel BREADY delay in ACLK cycles for this transaction (default = 0)
	data_beat_done[]	Write data channel beat ‘done’ flag array for this transaction
	transaction_done	Write transaction ‘done’ flag for this transaction
Returns	This Procedure returns a transaction ID, which identifies a unique transaction.	

Usage

```
--Write data burst length of 3 to start address 16.
trans = bfm.create_write_transaction(16, 3, tr_id, bfm_index,
      axi_tr_if_0(bfm_index));
trans.size = AXI_BYTES_4;
trans.data_words[0] = 'hACE0ACE1;
trans.data_words[1] = 'hACE2ACE3;
trans.data_words[2] = 'hACE4ACE5;
trans.data_words[3] = 'hACE6ACE7;
```

create_read_transaction()

This procedure creates a read transaction with a user-supplied start address and burst length. All other transaction fields default to legal protocol values, unless assigned values.

Table 8-5. VHDL AXI3 Master create_read_transaction Procedure

Prototype

```
procedure create_read_transaction
(
      addr            : in
      std_logic_vector(AXI_MAX_BIT_SIZE-1 downto
      0)|integer;
      burst_length    : in integer; --optional
      transaction_id  : out integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments	addr	Start address
	burst_length	Burst length (default = 0)
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"
Returns	This Procedure returns a transaction ID, which identifies a unique transaction.	
Protocol Transaction Fields	size	Burst size (defaults to width of bus) – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
	burst	Burst type – AXI_FIXED; AXI_INCR (default); AXI_WRAP; AXI_BURST_RSVD;
	lock	Burst lock – AXI_NORMAL (default); AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;

Table 8-5. VHDL AXI3 Master create_read_transaction Procedure

Protocol Transaction Fields (continued)	cache	Burst cache – AXI_NONCACHE_NONBUF (default); AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;
	prot	Burst protection (default) – AXI_NORM_SEC_DATA; AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
	id	Burst ID
	data_words[ ]	Data words array
	resp[ ]	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
Operational Transaction Fields	operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING (default);
	delay_mode	Delay mode - AXI_VALID2READY (default); AXI_TRANS2READY;
	address_valid_delay	Read address channel ARVALID delay in ACLK cycles for this transaction (default = 0)
	data_ready_delay[ ]	Read data channel RREADY delay array in ACLK cycles for this transaction (default = 0 for all elements)

Table 8-5. VHDL AXI3 Master create_read_transaction Procedure

data_beat_done[ ]	Read data channel beat ‘done’ flag array for this transaction
transaction_done	Read transaction ‘done’ flag for this transaction

Usage

```
--Read data burst length of 3 to start address 16.
create_read_transaction(16, 3, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

execute_transaction()

This procedure executes a master transaction in a blocking or nonblocking mode, based on the *operation_mode* configuration. For a write transaction, it auto-corrects the write strobe values if the transaction field *gen_write_strobes = 1*. On completion of this transaction, it sets the *transaction_done* flag to 1.

Table 8-6. VHDL AXI3 Master execute_transaction Procedure

Prototype

```
procedure execute_transaction
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id          : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments		
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
--By default it will execute a blocking transaction
execute_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

execute_write_addr_phase()

This procedure executes a master write address phase in a blocking or nonblocking mode, based on the *operation_mode* configuration.

Table 8-7. VHDL AXI3 Master execute_write_addr_phase Procedure

Prototype

```
procedure execute_write_addr_phase
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id          : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments		
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
--By default it will execute a blocking write address phase
execute_write_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

execute_read_addr_phase()

This procedure executes a master read address phase in a blocking or nonblocking mode, based on the *operation_mode* configuration.

Table 8-8. VHDL AXI3 Master execute_read_addr_phase Procedure

Prototype

```
procedure execute_read_addr_phase
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id          : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
--By default it will execute a blocking read address phase
execute_read_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

execute_write_data_burst()

This procedure executes a master write data burst in a blocking or nonblocking mode, based on the *operation_mode* configuration.

Table 8-9. VHDL AXI3 Master execute_write_data_burst Procedure

Prototype

```
procedure execute_write_data_burst
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id          : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
--By default it will execute a blocking write data burst
execute_write_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

execute_write_data_phase()

This procedure executes a master write data phase in a blocking or nonblocking mode, based on the *operation_mode* configuration.

VHDL AXI3 Master execute_write_data_phase Procedure

Prototype

```
procedure execute_write_data_phase
(
      transaction_id  : in integer;
      index           : in integer; --optional
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
--By default it will execute a single blocking write data phase
execute_write_data_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

get_read_data_burst()

This task waits for a read data burst and copies the data and response information via the *tr_if* signal interface.

Table 8-10. VHDL AXI3 Master get_read_data_burst Task

Prototype

```
procedure get_read_data_burst
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t; --
      optional
      signal tr_if    : inout
      axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"
Protocol Transaction Fields	data_words[]	Read data word array
	resp[]	Read response array
	id	Burst response ID
Operational Transaction Fields	data_beat_done[]	Read data beat array 'done' flags
	transaction_done	Read transaction 'done' flag for this transaction

Usage

```
get_read_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

get_read_data_phase()

This task waits for a read data phase(beat) and copies the data and response information via the *tr_if* signal interface. It sets the *RREADY* signal value at the appropriate time. On completion of a data phase(beat), this task sets the *data_beat_done[index]* to 1.

Table 8-11. VHDL AXI3 Master get_read_data_phase Task

Prototype

```
procedure get_read_data_phase
(
      transaction_id  : in integer;
      index           : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t;
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments	index	Array index for data beat array element
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"
Protocol Transaction Fields	data_words[index]	Read data word array element
	resp[index]	Read response array element
	id	Burst response ID
Operational Transaction Fields	data_beat_done[index]	Read data beat array element 'done' flag

Usage

```
get_read_data_phase(tr_id, data_index, bfm_index,
      axi_tr_if_0(bfm_index));
```

get_write_response_phase

This task waits for a write response phase and copies the response information via the *tr_if* signal interface. It sets the *BREADY* signal value at the appropriate time. It also sets the *transaction_done* flag to 1 on completion of the write response phase.

Table 8-12. VHDL AXI3 Master get_write_response_phase Procedure

Prototype

```
procedure get_write_response_phase
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"
Protocol Transaction Fields	resp	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
	id	Burst response ID
Operational Transaction Fields	transaction_done	Write transaction 'done' flag for this transaction

Usage

```
get_write_response_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

wait_on()

This tasks waits for an event on the *ACLK* or *ARESETn* signals.

Table 8-13. VHDL AXI3 Master wait_on Procedure

Prototype

```
procedure wait_on
(
      phase           : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

phase	Wait for - AXI_CLOCK_POSEDGE AXI_CLOCK_NEGEDGE AXI_CLOCK_ANYEDGE AXI_RESET_POSEDGE AXI_RESET_NEGEDGE AXI_RESET_ANYEDGE
bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
wait_on(AXI_RESET_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
wait_on(AXI_CLOCK_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
```

Chapter 9
VHDL AMBA AXI3 Slave BFM

The VHDL AMBA AXI3 Slave BFM implements the full AMBA AXI3 protocol. In addition to the standard protocol it supports user sideband signals *AWUSER* and *ARUSER*.

VHDL AXI3 Slave Timing

For detailed timing diagrams of the protocol bus activity refer to the AMBA AXI Protocol Specification, which you can use as the reference for the following details of the BFM and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising *ACLK* edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

VHDL AXI3 Slave Configuration

The VHDL AMBA AXI3 Slave BFM supports the full range of signals defined for the AMBA AXI3 protocol. You may customize the AMBA AXI3 Slave interface using the configuration fields described in Table 9-1.

Table 9-1. VHDL AXI3 Slave BFM Configuration

Configuration Field	Description
Signal Width #Parameter	
AXI_ADDRESS_WIDTH	Address width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
AXI_RDATA_WIDTH	Read data width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 1024.
AXI_WDATA_WIDTH	Write data width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 1024.
AXI_ID_WIDTH	ID width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
Timing Variables	
AXI_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.
AXI_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.

Table 9-1. VHDL AXI3 Slave BFM Configuration

AXI_CONFIG_MAX_ TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_ DATA_TRANSFER	The maximum number of write data beats that the AXI interface can generate as part of write data burst of write transfer. Default: 1024.
AXI_CONFIG_BURST_TIMEOUT_ FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
Master Attributes	
AXI_CONFIG_WRITE_CTRL_TO_ DATA_MINTIME	The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_WRITE_DATA_ TO_CTRL_MINTIME	The minimum delay between the start of a write data phase to the start of a write response phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_DEFAULT_ UNDER_RESET	The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true
Slave Attributes	
AXI_CONFIG_SUPPORT_ EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_ UNDER_RESET	The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
Error Detection	
AXI_CONFIG_ENABLE_ERRORS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to VHDL AXI3 Slave Timing for details of simulator time-steps.

VHDL AXI3 Slave Assertions

The slave BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

By default all built-in assertions are enabled in the slave BFM. To globally disable them in the slave BFM, use the set_config() command as the following example illustrates.

set_config(AXI_CONFIG_ENABLE_ERRORS,0,bfm_index,axi_tr_if_0(bfm_index));

Alternatively, you can disable individual built-in assertions by using a sequence of get_config() and set_config() commands on the respective assertion. For example, to disable assertion checking for the *AWLOCK* signal changing between the *AWVALID* and *AWREADY* handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI_CONFIG_ENABLE_ERROR,config_assert_bitvector,bfm_index,
axi_tr_if_0(bfm_index));

-- Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] := 0;

-- Set the new value of the assertion bit vector
set_config(AXI_CONFIG_ENABLE_ERROR,config_assert_bitvector,bfm_index,
axi_tr_if_0(bfm_index));
```

Note
Do not confuse the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector with the *AXI_CONFIG_ENABLE_ERRORS* global enable/disable.

To re-enable the *AXI_LOCK_CHANGED_BEFORE_AWREADY* assertion, following the above code sequence, assign the assertion within the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector to 1.

For a complete listing of assertions. refer to "AXI3 Assertions" on page 87.

VHDL AXI3 Slave Application Program Interface

This section describes the SystemVerilog AMBA® AXI3™ Slave API for the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the AXI3 Slave BFM.

Table 9-2. VHDL AXI3 Slave set_config() Procedure

Prototype

```
procedure set_config
(
   config_name    : in std_logic_vector(7 downto 0);
   config_val     : in std_logic_vector(AXI_MAX_BIT_SIZE-1
   downto 0)|integer;
   bfm_id         : in integer;
   path_id        : in axi_path_t; --optional
   signal tr_if   : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments **config_name**

AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "VHDL AXI3 Slave BFM Configuration" on page 145 for descriptions and valid values.

config_value

bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Returns

None

Usage

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi_tr_if_0(bfm_index));
Set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi_tr_if_0(bfm_index));
```

get_config()

This function gets the configuration of the AXI3 Slave BFM.

Table 9-3. VHDL AXI3 Slave get_config() Procedure

Prototype

```
procedure get_config
     (
     config_name    : in std_logic_vector(7 downto 0);
     config_val     : in std_logic_vector(AXI_MAX_BIT_SIZE-1
     downto 0)|integer;
     bfm_id         : in integer;
     path_id        : in axi_path_t; --optional
     signal tr_if   : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

config_name	
AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "VHDL AXI3 Slave BFM Configuration" on page 145 for descriptions and valid values.

config_value	
bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Returns

config_val

Usage

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value, bfm_index,
      axi_tr_if_0(bfm_index));
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
      axi_tr_if_0(bfm_index));
```

create_slave_transaction()

This function creates a slave transaction. All other transaction fields default to legal protocol values, unless assigned values.

Table 9-4. VHDL AXI3 Slave create_slave_transaction Function

Prototype

```
procedure create_slave_transaction
(
  transaction_id  : out integer;
  bfm_id          : in integer;
  path_id         : in axi_path_t; --optional
  signal tr_if    : inout axi_vhd_if_struct_t
);
```

Arguments	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"
Operational Transaction Fields	delay	Address channel phase A*READY delay in ACLK cycles for this transaction
	delay_mode	Delay mode - AXI_VALID2READY (default); AXI_TRANS2READY;
	operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING (default);
	data_valid_delay[]	Read data channel RVALID delay array in ACLK cycles for this transaction (default = 0 for all elements)
	data_ready_delay[]	Write data channel WREADY delay array in ACLK cycles for this transaction (default = 0 for all elements)
	write_response_valid_delay	Write response channel BVALID delay in ACLK cycles for this transaction (default = 0)

Usage

```
// Create a slave read transaction
create_slave_transaction(read_trans, bfm_index, AXI_PATH_3,
      axi_tr_if_3(bfm_index));
// Create a slave write transaction
create_slave_transaction(write_trans, bfm_index, axi_tr_if_0(bfm_index));
```

execute_read_data_burst()

This procedure executes a slave read data burst in a blocking (default) or nonblocking mode, based on the *operation_mode* configuration. On completion of this transaction it sets the *transaction_done* flag to 1.

Table 9-5. VHDL AXI3 Slave execute_read_data_burst Procedure

Prototype

```
procedure execute_read_data_burst
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi_path_t; --optional
  signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments		
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
execute_read_data_burst(read_trans, bfm_index, AXI_PATH_4,
axi_tr_if_4(bfm_index));
```

execute_read_data_phase()

This procedure executes a slave read data phase(beat) in blocking (default) or nonblocking mode. On completion of one data phase(beat), this task sets the *data_beat_done[index]* array element to 1.

Table 9-6. VHDL AXI3 Slave execute_read_data_phase Procedure

Prototype

```
procedure execute_read_data_phase
(
  transaction_id  : in integer;
  index           : in integer;  --optional
  bfm_id          : in integer;
  path_id         : in axi_path_t; --optional
  signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

index	Array index for data beat array element.
transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section “Overloaded Procedure Common Arguments”.

Usage

```
execute_read_data_phase(read_trans, element_index, bfm_index, AXI_PATH_4,
      axi_tr_if_4(bfm_index));
```

execute_write_response_phase()

This procedure executes a slave write response phase in blocking (default) or nonblocking mode. It sets the *transaction_done* flag to 1 on completion.

Table 9-7. VHDL AXI3 Slave execute_write_response_phase Procedure

Prototype

```
procedure execute_write_response_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi_path_t; --optional
  signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments		
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments".

Usage

```
execute_write_response_phase(write_trans, bfm_index, AXI_PATH_2,
      axi_tr_if_2(bfm_index));
```

get_write_addr_phase()

This procedure gets a write address phase, copying the information va the *tr_if* argument.

Table 9-8. VHDL AXI3 Slave get_write_addr_phase Procedure

Prototype

```
procedure get_write_addr_phase
(
  transaction_id  : in integer;
  bfm_id          : in integer;
  path_id         : in axi_path_t;
  signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments		
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments".

Usage

```
get_write_addr_phase(write_trans, bfm_index, axi_tr_if_0(bfm_index));
```

get_read_addr_phase()

This procedure gets a read address phase and copies the address information using the *tr_if* signal argument.

Table 9-9. VHDL AXI3 Slave get_read_addr_phase Procedure

Prototype

```
procedure get_read_addr_phase
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments		
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments".

Usage

```
get_read_addr_phase(read_trans, bfm_index, AXI_PATH_3,
   axi_tr_if_3(bfm_index));
```

get_write_data_phase()

This procedure gets a write data phase(beat) and copies the data information using the *tr_if* signal argument. On completion of a single data phase, it sets the *data_beat_done[index]* array element to 1.

Table 9-10. VHDL AXI3 Slave get_write_data_phase Procedure

Prototype

```
procedure get_write_data_phase
(
  transaction_id  : in integer;
  index           : in integer; --optional
  last            : out integer;
  bfm_id          : in integer;
  path_id         : in axi_path_t; -optional
  signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

index	Array index for data beat array element
last	Last write data beat signal WLAST.
trans bfm_id path_id (optional) tr_if	For descriptions, refer to the section “Overloaded Procedure Common Arguments”.

Usage

```
get_write_data_phase(write_trans, data_index, last, bfm_index,
      AXI_PATH_1, axi_tr_if_1(bfm_index));
```

get_write_data_burst()

This task gets a write data burst and copies the data information using the *tr_if* signal argument.

Table 9-11. VHDL AXI3 Slave get_write_data_burst Procedure

Prototype

```
procedure get_write_data_burst
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t;
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments		
	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments".

Usage

```
get_write_data_burst(write_trans, bfm_index, AXI_PATH_1,
      axi_tr_if_1(bfm_index));
```

wait_on()

This tasks waits for an event on the *ACLK* or *ARESETn* signals.

Table 9-12. VHDL AXI3 Slave wait_on Procedure

Prototype

```
procedure wait_on
(
      phase           : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

phase	Wait for - AXI_CLOCK_POSEDGE AXI_CLOCK_NEGEDGE AXI_CLOCK_ANYEDGE AXI_RESET_POSEDGE AXI_RESET_NEGEDGE AXI_RESET_ANYEDGE
bfm_id path_id (optional) tr_if	For descriptions, refer to the section "Overloaded Procedure Common Arguments"

Usage

```
wait_on(AXI_RESET_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
wait_on(AXI_CLOCK_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
```

Helper Functions

AMBA AXI protocols typically provide a start address only within a transaction, with the following addresses for each byte of a data burst calculated using the size, length, and type transaction fields of the transaction. Helper functions provide you with a simple interface to set and get actual address/data values.

get_write_addr_data()

This helper function provides the address and data content of the memory for the address *index* and data *byte_index* numbers.

Table 9-13. VHDL AXI3 Slave get_write_addr_data Procedure

Prototype

```
procedure get_write_addr_data
(
      transaction_id  : in integer;
      index           : in integer;
      byte_index      : in integer;
      dynamic_size    : out integer;
      addr            : out
      std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
      data            : out std_logic_vector(7 downto 0);
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Arguments

index	Address array element number
byte_index	Data byte array element number
dynamic_size	Size of data array
addr	Write address
data	Write data byte
transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section “Overloaded Procedure Common Arguments”

Usage

```
get_write_addr_data(write_trans, data_index, byte_index, byte_length,
      addr, data, bfm_index, AXI_PATH_1, axi_tr_if_1(bfm_index));
```

get_read_addr()

This helper function provides the address of the memory for the address *index* and data *byte_index* numbers.

Table 9-14. VHDL AXI3Slave get_read_addr Procedure

Prototype

```
procedure get_read_addr
(
      transaction_id  : in integer;
      index           : in integer;
      byte_index      : in integer;
      dynamic_size    : out integer;
      addr            : out
      std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Arguments

index	Address array element number
byte_index	Data byte array element number
dynamic_size	Size of data array
addr	Write address
transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section “Overloaded Procedure Common Arguments”

Usage

```
get_read_addr(read_trans, data_index, byte_index, byte_length, addr,
      bfm_index, AXI_PATH_1, axi_tr_if_1(bfm_index));
```

set_read_data()

This helper function sets read data content of the transaction from memory for the address *index* and data *byte_index* arguments.

Table 9-15. VHDL AXI3Slave get_write_addr_data Procedure

Prototype

```
procedure get_write_addr_data
(
      transaction_id  : in integer;
      index           : in integer;
      byte_index      : in integer;
      dynamic_size    : out integer;
      addr            : out
      std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
      data           : out std_logic_vector(7 downto 0);
      bfm_id          : in integer;
      path_id         : in axi_path_t; --optional
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Arguments

index	Address array element number
byte_index	Data byte array element number
dynamic_size	Size of data array
addr	Write address
data	Write data byte
transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section “Overloaded Procedure Common Arguments”

Usage

```
set_read_data(read_trans, data_index, byte_index, byte_length, addr, data,
      bfm_index, AXI_PATH_4, axi_tr_if_4(bfm_index));
```

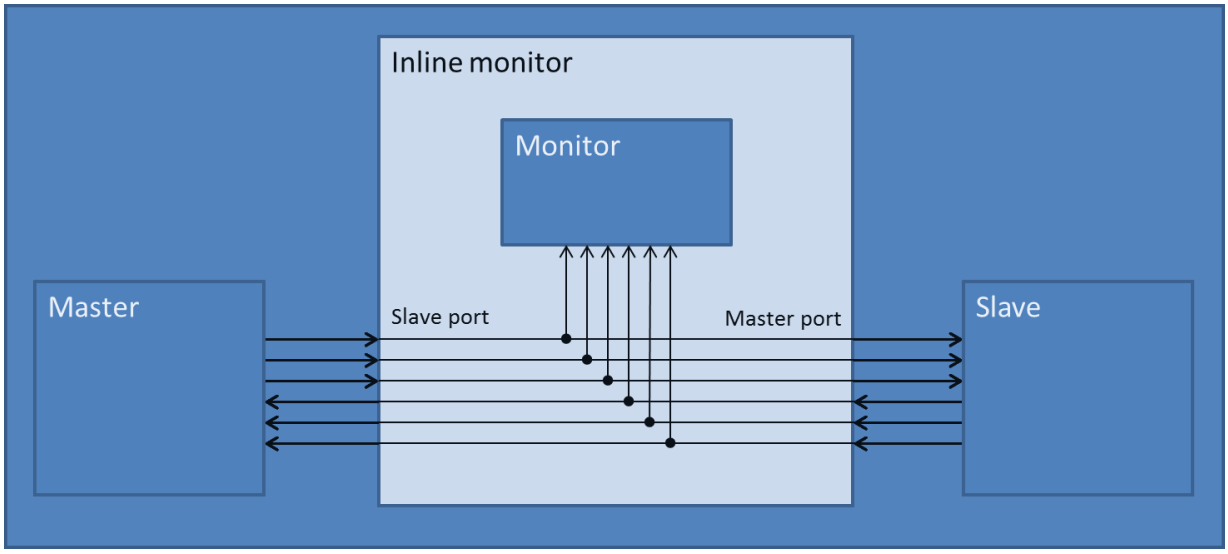
Chapter 10
VHDL AMBA AXI3 Monitor BFM

The VHDL AMBA AXI3 Monitor BFM implements the full AMBA AXI3 protocol. In addition to the standard protocol it supports user sideband signals *AWUSER* and *ARUSER*.

VHDL AXI3 Inline Monitor Connection

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped within an inline monitor interface and connected inline, between a master and slave, as shown in Figure 10-1. It has separate master and slave ports and monitors protocol traffic between a master and slave. By construction, the monitor has access to all the facilities provided by the monitor BFM.

Figure 10-1. Inline Monitor Connection Diagram



VHDL AXI3 Monitor Timing

For detailed timing diagrams of the protocol bus activity refer to the AMBA AXI3 protocol specification, which you can use as the reference for the following details of the BFM and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising *ACLK* edges. Therefore, the slave BFM does not contain any timescale,

timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

VHDL AXI3 Monitor Configuration

The AXI3 Monitor BFM supports the full range of signals defined for the AMBA AXI3 protocol. You may customize the AXI3 Monitor BFM using the configuration fields described in Table 10-1

Table 10-1. AXI3 Monitor BFM Configuration

Configuration Field	Description
Signal Width #Parameter	
AXI_ADDRESS_WIDTH	Address width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
AXI_RDATA_WIDTH	Read data width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 1024.
AXI_WDATA_WIDTH	Write data width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 1024.
AXI_ID_WIDTH	ID width in bits. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
Timing Variables	
AXI_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.
AXI_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals[1]. Default: 0.
AXI_CONFIG_MAX_ TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_ DATA_TRANSFER	The maximum number of write data beats that the AXI interface can generate as part of write data burst of write transfer. Default: 1024.
AXI_CONFIG_BURST_TIMEOUT_ FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
Master Attributes	
AXI_CONFIG_WRITE_CTRL_TO_ DATA_MINTIME	The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_WRITE_DATA_ TO_CTRL_MINTIME	The minimum delay between the start of a write data phase to the start of a write response phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_DEFAULT_ UNDER_RESET	The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true

Table 10-1. AXI3 Monitor BFM Configuration

Slave Attributes	
AXI_CONFIG_SUPPORT_ EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_ UNDER_RESET	The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
Error Detection	
AXI_CONFIG_ENABLE_ERRORS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to VHDL AXI3 Monitor Timing for details of simulator time-steps.

AXI3 Monitor Assertions

The monitor BFM performs protocol error checking via built-in assertions.

Note

The built-in BFM assertions are independent of programming language and simulator.

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the set_config() command as the following example illustrates:

```
set_config(AXI_CONFIG_ENABLE_ERRORS,0,bfm_index, axi_tr_if_0(bfm_index));
```

Alternatively, you a disable individual built-in assertions by using a sequence of get_config() and set_config() commands on the respective assertion. For example, to disable assertion checking for the *AWLOCK* signal changing between the *AWVALID* and *AWREADY* handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI_CONFIG_ENABLE_ERROR,config_assert_bitvector, bfm_index,
axi_tr_if_0(bfm_index));

-- Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] := 0;

-- Set the new value of the assertion bit vector
```

```
set_config(AXI_CONFIG_ENABLE_ERROR,config_assert_bitvector, bfm_index,
axi_tr_if_0(bfm_index));
```

Note

Do not confuse the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector with the *AXI_CONFIG_ENABLE_ERRORS* global enable/disable.

To re-enable the *AXI_LOCK_CHANGED_BEFORE_AWREADY* assertion, following the above code sequence, assign the assertion within the *AXI_CONFIG_ENABLE_ERROR* assertion bit vector to 1.

For a complete listing of assertions, refer to "AXI3 Assertions" on page 87.

VHDL AXI3 Monitor API

This section describes the VHDL AMBA AXI3 Monitor API for the Mentor Verification IP Altera Edition.

set_config()

This function sets the configuration of the AXI3 Monitor BFM.

Table 10-2. VHDL AXI3 Monitor set_config() Procedure

Prototype

```
procedure set_config
(
       config_name   : in std_logic_vector(7 downto 0);
       config_val    : in std_logic_vector(AXI_MAX_BIT_SIZE-1
          downto 0)|integer;
       bfm_id        : in integer;
       path_id       : in axi_path_t; --optional
       signal tr_if  : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

config_name	
AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "VHDL AXI3 Monitor Configuration" on page 164 for descriptions and valid values.

config_value	
bfm_id path_id (optional) tr_if	For descriptions, refer to the section on "Overloaded Procedure Common Arguments"

Returns

None

Usage

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
   axi_tr_if_0(bfm_index));
Set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
   axi_tr_if_0(bfm_index));
```

get_config()

This procedure gets the configuration of the BFM.

Table 10-3. VHDL AXI3 Monitor get_config() Procedure

Prototype

```
procedure get_config
(
       config_name   : in std_logic_vector(7 downto 0);
       config_val    : in std_logic_vector(AXI_MAX_BIT_SIZE-1
           downto 0)|integer;
       bfm_id        : in integer;
       path_id       : in axi_path_t; --optional
       signal tr_if  : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

config_name

AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET AXI_CONFIG_ENABLE_ERROR AXI_CONFIG_ENABLE_ERRORS AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Refer to "VHDL AXI3 Monitor Configuration" on page 164 for descriptions and valid values.

config_value

bfm_id path_id (optional) tr_if	For descriptions, refer to the section on "Overloaded Procedure Common Arguments"

Returns

config_value

Usage

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value,
   bfm_index, axi_tr_if_0(bfm_index));
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, config_value,
   bfm_index, axi_tr_if_0(bfm_index));
```

create_monitor_transaction()

This procedure creates a transaction to monitor protocol transfers.

Table 10-4. VHDL AXI3 Monitor create_monitor_transaction Procedure

Prototype

```
procedure create_monitor_transaction
(
   transaction_id  : out integer;
   bfm_id          : in integer;
   path_id         : in axi_path_t; --optional
   signal tr_if    : inout axi_vhd_if_struct_t
);
```

Arguments

transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section on "Overloaded Procedure Common Arguments"

Usage

```
// Create a monitor transaction
create_monitor_transaction(trans, bfm_index, axi_tr_if_0(bfm_index));
```

get_rw_transaction()

This procedure waits for a complete read/write transaction and then gets the complete transaction content via the *tr_if* signal argument.

Table 10-5. VHDL AXI3 Monitor get_rw_transaction Procedure

Prototype

```
procedure get_rw_transaction
(
      transaction_id  : in integer;
      bfm_id          : in integer;
      path_id         : in axi_path_t;
      signal tr_if    : inout axi_vhd_if_struct_t
);
```

Arguments	transaction_id bfm_id path_id (optional) tr_if	For descriptions, refer to the section on "Overloaded Procedure Common Arguments"
Protocol Transaction Fields	addr burst_length	Start address Burst length
	size	Burst size – AXI_BYTES_1; AXI_BYTES_2; AXI_BYTES_4; AXI_BYTES_8; AXI_BYTES_16; AXI_BYTES_32; AXI_BYTES_64; AXI_BYTES_128;
	burst	Burst type – AXI_FIXED; AXI_INCR; AXI_WRAP; AXI_BURST_RSVD;

Table 10-5. VHDL AXI3 Monitor get_rw_transaction Procedure

Protocol Transaction Fields	lock	Burst lock – AXI_NORMAL; AXI_EXCLUSIVE; AXI_LOCKED; AXI_LOCK_RSVD;
	cache	Burst cache – AXI_NONCACHE_NONBUF; AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;
	prot	Burst protection – AXI_NORM_SEC_DATA; AXI_PRIV_SEC_DATA; AXI_NORM_NONSEC_DATA; AXI_PRIV_NONSEC_DATA; AXI_NORM_SEC_INST; AXI_PRIV_SEC_INST; AXI_NORM_NONSEC_INST; AXI_PRIV_NONSEC_INST;
	id	Burst ID
	data_words[]	Data words array
	write_strobes[]	Write strobes array – Each element 0 or 1
	resp	Burst response – AXI_OKAY; AXI_EXOKAY; AXI_SLVERR; AXI_DECERR;
	addr_user	Address channel user data
	read_or_write	Read or write flag – AXI_TRANS_READ; AXI_TRANS_WRITE;

Table 10-5. VHDL AXI3 Monitor get_rw_transaction Procedure

Operational Transaction Fields

gen_write_strobes	Generate write strobes flag – 0 = user supplied write strobes 1 = auto-generated write strobes
operation_mode	Operation mode – AXI_TRANSACTION_NON_BLOCKING; AXI_TRANSACTION_BLOCKING;
delay_mode	AXI_VALID2READY; AXI_TRANS2READY;
write_data_mode	AXI_DATA_AFTER_ADDRESS; AXI_DATA_WITH_ADDRESS;
address_valid_delay	Write address channel AWVALID delay in ACLK cycles for this transaction
data_valid_delay[]	Write data channel WVALID delay array in ACLK cycles for this transaction
write_response_ ready_delay	Write response channel BREADY delay in ACLK cycles for this transaction
data_beat_done[]	Write data channel beat ‘done’ flag array for this transaction
transaction_done	Write transaction ‘done’ flag for this transaction

Usage

```
get_rw_transaction(trans, bfm_index, axi_tr_if_0(bfm_index));
```

wait_on()

This tasks waits for an event on the *ACLK* or *ARESETn* signals.

Table 10-6. VHDL AXI3 Monitor wait_on Procedure

Prototype

```
procedure wait_on
(
      phase            : in integer;
      bfm_id           : in integer;
      path_id          : in axi_path_t; --optional
      signal tr_if     : inout axi_vhd_if_struct_t
);
```

Transaction Variables

Arguments

phase	Wait for - AXI_CLOCK_POSEDGE AXI_CLOCK_NEGEDGE AXI_CLOCK_ANYEDGE AXI_RESET_POSEDGE AXI_RESET_NEGEDGE AXI_RESET_ANYEDGE
bfm_id path_id (optional) tr_if	For descriptions, refer to the section on "Overloaded Procedure Common Arguments"

Usage

```
wait_on(AXI_RESET_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
wait_on(AXI_CLOCK_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
```

Chapter 11
VHDL Tutorials

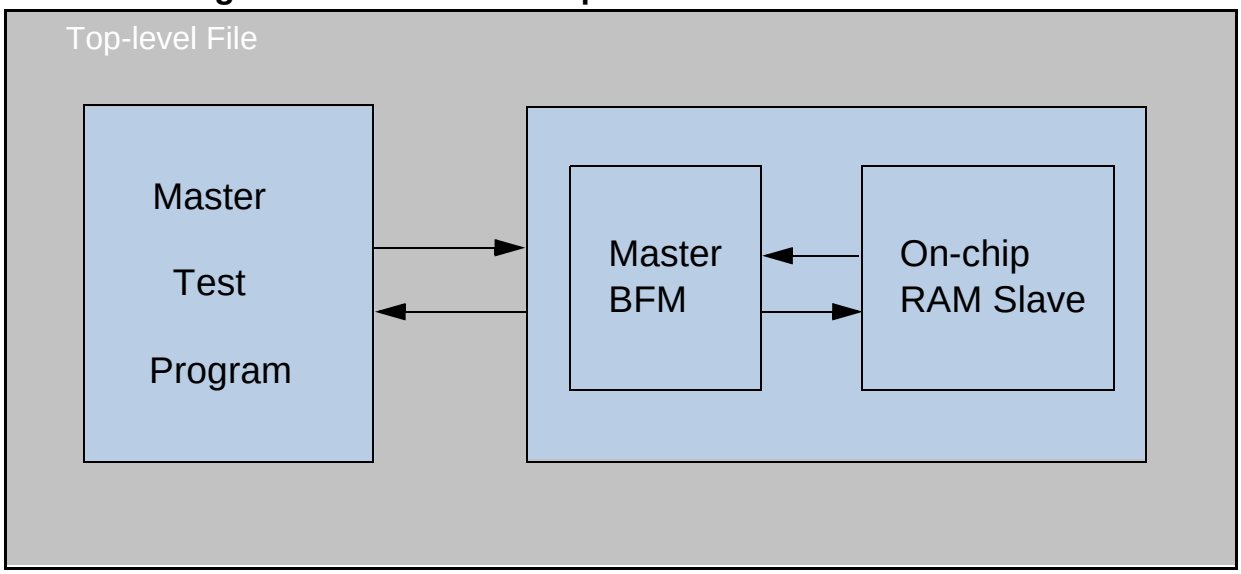
This chapter discusses how to use the Mentor Verification IP Altera Edition Master and Slave BFMs to verify a master and a slave component. The first example, the slave DUT, is an on-chip RAM that complies with the AXI3 protocol. An AXI3 Master BFM and Master Test Program verify the behavior of the slave DUT. The second example verifies an AXI3 master DUT using an AXI3 Slave BFM and Slave Test Program.

Following this top-level discussion of how you verify a master and a slave component using the Mentor Verification IP Altera Edition is a brief example of how to run Qsys, the powerful system integration tool in the Quartus II software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details on this example, refer to "Getting Started with Qsys and the BFMs" on page 191.

Verifying a VHDL Slave DUT

In this example the DUT is an on-chip RAM slave component connected to a Master BFM at the signal-level. A Master Test Program, written at the transaction-level, generates stimulus via the Master BFM to the slave DUT. Figure 11-1 illustrates a typical top-level testbench environment that verifies the slave component.

Figure 11-1. Slave DUT Top-level Testbench Environment



In this example an AXI3 slave DUT component is connected to an AXI3 Master BFM at the signal-level. The AXI3 Master Test Program configures the AXI3 Master BFM and instructs it to create and execute transactions to the slave DUT. The Master Test Program compares data

written to and read back from the slave DUT and reports any mismatches. The test passes if the read data is correct compared with the written data.

A top-level file instantiates and connects all the components required to test and monitor the DUT. This file also controls the system clock and reset.

VHDL Master Test Program

Using the VHDL AXI3 Master BFM API, this VHDL Master Test Program creates a wide range of stimulus scenarios which test the slave DUT. This tutorial restricts the stimulus to a write transaction followed by a read transaction to the same address, which then compares the read data with the previously written data. For a complete code example of this VHDL Master Test Program, refer to "VHDL Master BFM Code Example" on page 211 in Appendix B.

The code excerpt in Example 11-1 shows the Master Test Program architecture definition *master_test_program_a*. It defines two variables, *tr_id* and *data_words*, to hold the transaction identifier number and the data word used for writing and reading, respectively. An additional system clock cycle is waited on after reset to satisfy the AXI3 protocol requirement specified in Section 11.1.2 of the AMBA AXI Protocol Specification before executing transactions.

Example 11-1. Architecture Definition and Initialization

```
architecture master_test_program_a of master_test_program is
begin

  -- Master test
  process
    variable tr_id: integer;
    variable data_words :  std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
  begin
    wait_on(AXI_RESET_POSEDGE, index, axi_tr_if_0(index));
    wait_on(AXI_CLOCK_POSEDGE, index, axi_tr_if_0(index));
```

To generate AXI3 protocol traffic, the Master Test Program must create the transaction before executing it. The code excerpt in Example 11-2 uses the AMBA AXI3 Master BFM *create_write_transaction()* function to provide only the start address argument of the transaction to be created. Because this function call provides only the start address, the burst-length argument automatically defaults to a value of zero—indicating a burst length of a single beat (refer to "AMBA AXI3 Master BFM" on page 123 for more details).

This example has an AXI3 data bus width of 32-bits; therefore a single beat of data conveys 4-bytes across the data bus. The *set_data_words()* function sets the '*data_words[0]*' transaction field with the value of 1 on byte lane 1, resulting in a value of 32'h0000_0100. However, the AXI3 protocol permits narrow transfers with the use of the write strobes signal *WSTRB* to indicate which byte lane contains valid write data. Similarly, you can use the *set_write_strobes()* function to set the *write_strobes[0]* transaction field with the value of

4'b0010, indicating that only valid data is being transferred on byte lane 1. The write transaction then executes on the protocol signals using the"execute_transaction()" on page 35.

All other transaction fields default to legal protocol values for more details).

Example 11-2. Write Transaction Creation and Execution

```
-- Write data value 1 on byte lanes 1 to address 1.
create_write_transaction(1, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"00000100";
set_data_words(data_words, tr_id, index, axi_tr_if_0(index));
set_write_strobes(2, tr_id, index, axi_tr_if_0(index));
report "master_test_program: Writing data (1) to address (1)";

-- By default it will run in Blocking mode
execute_transaction(tr_id, index, axi_tr_if_0(index));
```

The code excerpt in Example 11-3 uses the "VHDL AXI3 Master BFM" to read the data that has just been written into the slave memory. The Master Test Program first creates a read transaction using the *create_read_transaction()* function and provides only the start address argument of the transaction to be created. Because this function call provides only the start address, the burst-length automatically defaults to a value of zero—indicating a burst length of a single beat (refer to "VHDL AXI3 Master BFM" for more details).

The *set_id()* function is then used to set the transaction '*id*' transaction field to be 1 before executing the read transaction onto the protocol signals.

The read data is obtained using the *get_data_words()* function to get the '*data_words*' transaction field value. The result of the read data is compared with the expected data—and a report message displays the result.

Example 11-3. Read Transaction Creation and Execution

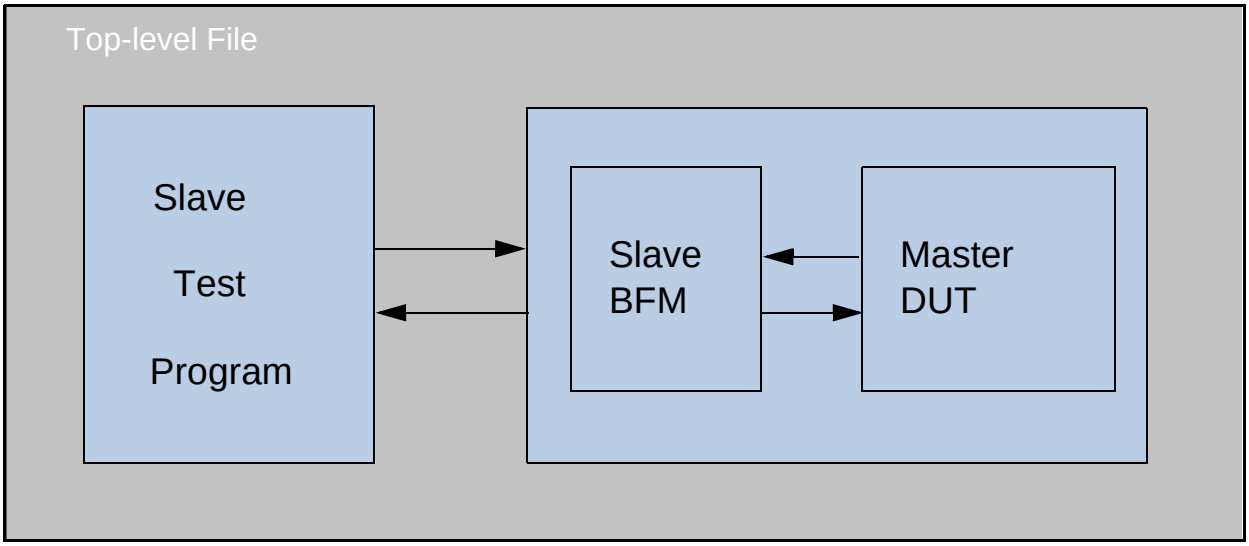
```
--Read data from address 1.
create_read_transaction(1, tr_id, index, axi_tr_if_0(index));
set_id(1, tr_id, index, axi_tr_if_0(index));
execute_transaction(tr_id, index, axi_tr_if_0(index));

get_data_words(data_words, tr_id, index, axi_tr_if_0(index));
if(data_words(31 downto 0) = x"00000100") then
      report "master_test_program: Read correct data (1) at address (1)";
else
      report "master_test_program: Expected data (1) at address 1, but got
      " & integer'image(to_integer(data_words(31 downto 0)));
end if;
```

Verifying a VHDL Master DUT

In this example the DUT is a master component connected to a Slave BFM at the signal-level. A Slave Test Program, written at the transaction-level, generates response stimulus via the Slave BFM to the Master DUT. Figure 11-2 illustrates a typical top-level testbench environment to verify a master component.

Figure 11-2. Master DUT Top-level Testbench Environment



In this tutorial example, a VHDL AXI3 master DUT component is connected to a VHDL AXI3 Slave BFM at the signal-level. The VHDL AXI3 Slave Test Program configures the VHDL AXI3 Slave BFM and instructs it to create and respond to transactions from the master DUT, storing the write data into an internal memory location.

A top-level file instantiates and connects all the components required to test and monitor the DUT. This program also controls the system clock and reset.

VHDL Slave Test Program

The Slave Test Program is a memory model which contains two APIs: a "VHDL Basic Slave API Definition" and a "VHDL Advanced Slave API Definition".

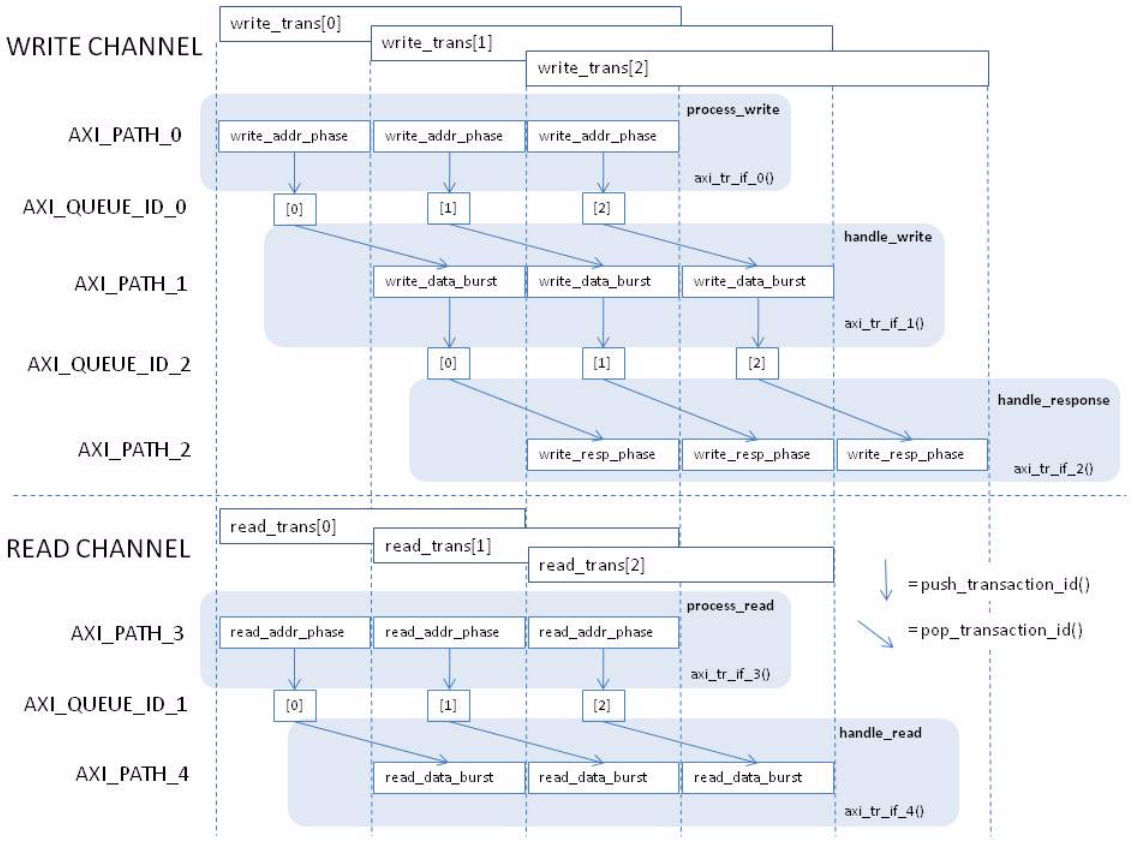
The *VHDL Basic Slave API Definition* allows you to create a wide range of stimulus scenarios to test a master DUT. This simple API design illustrates the creation of slave stimulus based on the AXI3 default response of *OKAY* to master read and write transactions.

The *VHDL Advanced Slave API Definition* allows you to create additional response scenarios to

AXI3 transactions. For example, a successful exclusive transaction requires an *EXOKAY* response. Figure 11-3 illustrates API pipelining with multiple write and read transactions.

For a complete code listing of the Slave Test Program APIs, refer to the "VHDL Slave BFM Code Example" on page 216 in Appendix B.

Figure 11-3. API Pipelining



VHDL Basic Slave API Definition

The Slave Test Program Basic API contains:

- Procedures that read and write a byte of data to *internal memory do_byte_read()* and *do_byte_write()*, respectively.
- Procedures to configure the AXI3 protocol channel handshake delays *set_read_address_ready_delay()*, set_write_address_ready_delay(), *set_write_data_ready_delay()*, *set_read_data_valid_delay()* and *set_wr_resp_valid_delay()*.
- Procedures to process read and write transactions, *process_read* and *process write*, respectively. If you need to create other responses, i.e. EXOKAY, DECERR, or SLVERR, then you will need to edit these procedures to provide the required response.
- A *slave_mode* transaction field to control the behavior of reading and writing to the internal memory.

The internal memory for the slave is defined as an array of 8-bits, so that each byte of data is stored as an address/data pair.

Example 11-4. internal memory

```
type memory_t is array (0 to 2**16-1) of std_logic_vector(7 downto 0);
shared variable mem : memory_t;
```

The *do_byte_read()* procedure when called will read a *data* byte from the internal memory *mem* given an address location *addr*, as demonstrated in Example 11-5.

You may edit this procedure to modify the way the read data is extracted from the internal memory.

Example 11-5. do_byte_read()

```
-- Procedure : do_byte_read
-- Procedure to provide read data byte from memory at particular input
-- address
procedure do_byte_read
(
      addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
      data : out std_logic_vector(7 downto 0)
) is
begin
      data := mem(to_integer(addr));
end do_byte_read;
```

The *do_byte_write()* procedure when called writes a *data* byte to the internal memory *mem* given an address location *addr*, as Example 11-6 illustrates.

You may edit this procedure to modify the way the write data is stored in the internal memory.

Example 11-6. do_byte_write()

```
-- Procedure : do_byte_write
-- Procedure to write data byte to memory at particular input address
procedure do_byte_write
(
      addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
      data : in std_logic_vector(7 downto 0)
) is
begin
      mem(to_integer(addr)) := data;
end do_byte_write;
```

The *set_read_address_ready_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called it configures the ARREADY handshake signal to be delayed by a number of ACLK cycles which extends the length of the read address phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to *"Delay Mode"* on page 20 for details). Example 11-7 demonstrates setting the ARREADY signal delay by 4 ACLK cycles.

You may edit this procedure to change the ARREADY signal delay.

Example 11-7. set_read_address_ready_delay()

```
-- Procedure : set_read_address_ready_delay
-- This is used to set read address phase ready delay to extend phase
procedure set_read_address_ready_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(4, id, index, tr_if);
end set_read_address_ready_delay;

procedure set_read_address_ready_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(4, id, index, path_id, tr_if);
end set_read_address_ready_delay;
```

The *set_write_address_ready_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called it configures the *AWREADY* handshake signal to be delayed by a number of *ACLK* cycles, which extends the length of the write address phase. The starting point of the delay is determined by the configuration of the

delay_mode operational transaction field (refer to "Delay Mode" on page 20 for details). Example 11-8 demonstrates setting the *AWREADY* signal delay by 2 *ACLK* cycles.

You may edit this procedure to change the AWREADY signal delay.

Example 11-8. set_write_address_ready_delay()

```
-- Procedure : set_write_address_ready_delay
-- This is used to set write address phase ready delay to extend phase
procedure set_write_address_ready_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(2, id, index, tr_if);
end set_write_address_ready_delay;

procedure set_write_address_ready_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(2, id, index, path_id, tr_if);
end set_write_address_ready_delay;
```

The *set_write_data_ready_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called it configures the *WREADY* signal handshake to be delayed by a number of *ACLK* cycles which extends the length of each write data phase (beat) within a write data burst. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "Delay Mode" on page 20 for details).

For each write data phase (beat) the delay value of the WREADY signal is stored in an element of the *data_ready_delay[]* array for the transaction, as demonstrated in Example 11-9.

You may edit this procedure to change the WREADY signal delays.

Example 11-9. set_write_data_ready_delay()

```
-- Procedure : set_write_data_ready_delay
-- This will set the ready delays for each write data phase in a write data
-- burst
procedure set_write_data_ready_delay
(
      id : integer;
      signal tr_if : inout axi_vhd_if_struct_t
) is
      variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, tr_if);
    for i in  0 to burst_length loop
```

```
      set_data_ready_delay(i, i, id, index, tr_if);
    end loop;
end set_write_data_ready_delay;

procedure set_write_data_ready_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
    variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, path_id, tr_if);
    for i in  0 to burst_length loop
      set_data_ready_delay(i, i, id, index, path_id, tr_if);
    end loop;
end set_write_data_ready_delay;
```

The set_read_data_valid_delay() procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called it configures the *RVALID* signal to be delayed by a number of *ACLK* cycles with the effect of delaying the start of each read data phase (beat) within a read data burst. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field ("Delay Mode" on page 20 or details).

For each read data phase (beat) the delay value of the *RVALID* signal is stored in an element of the *data_valid_delay[]* array for the transaction, as demonstrated in Example 11-10.

You may edit this procedure to change the *RVALID* signals delays.

Example 11-10. set_read_data_valid_delay()

```
-- Procedure : set_read_data_valid_delay
-- This will set the ready delays for each write data phase in a write data
-- burst
procedure set_read_data_valid_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
    variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, tr_if);
    for i in  0 to burst_length loop
      set_data_valid_delay(i, i, id, index, tr_if);
    end loop;
end set_read_data_valid_delay;

procedure set_read_data_valid_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, path_id, tr_if);
```

```
    for i in  0 to burst_length loop
      set_data_valid_delay(i, i, id, index, path_id, tr_if);
    end loop;
end set_read_data_valid_delay;
```

The *set_wr_resp_valid_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called, it configures the *BREADY* signal handshake to be delayed by a number of *ACLK* cycles, which extends the length of the write response phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "Delay Mode" on page 20 for details). Example 11-11 demonstrates setting the *BREADY* signal delay by 2 *ACLK* cycles.

You can edit this procedure to change the *BREADY* signal delay.

Example 11-11. set_wr_resp_valid_delay()

```
-- Procedure : set_wr_resp_valid_delay
-- This is used to set write response phase valid delay to start driving
-- write response phase after specified delay.
procedure set_wr_resp_valid_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
begin
    set_write_response_valid_delay(0, id, index, tr_if);
end set_wr_resp_valid_delay;

procedure set_wr_resp_valid_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
begin
    set_write_response_valid_delay(0, id, index, path_id, tr_if);
end set_wr_resp_valid_delay;
```

There is a *slave_mode* transaction field that you can configure to control the behavior of reading and writing to the *internal memory*. It has two modes: *AXI_TRANSACTION_SLAVE* and *AXI_PHASE_SLAVE* (refer to Example 11-12).

Example 11-12. slave_mode

```
-- Slave mode type definition
type axi_slave_mode_e is (AXI_TRANSACTION_SLAVE, AXI_PHASE_SLAVE);

-- Slave mode selection : Default is transaction-level slave
signal slave_mode : axi_slave_mode_e := AXI_TRANSACTION_SLAVE;
```

The default *AXI_TRANSACTION_SLAVE* mode "saves up" an entire data burst and modifies the Slave Test Program's internal memory in zero time for the whole burst. Therefore, a read from

internal memory is buffered at the beginning of the read burst for the whole burst. The buffered read data is then transmitted over the protocol signals to the master on a phase-by-phase (beat-by-beat) basis. For a write, the write data burst is buffered on a phase-by-phase (beat-by-beat) basis for the whole burst. Only at the end of the write burst are the buffered contents written to the internal memory.

The *AXI_PHASE_SLAVE* mode changes the Slave Test Program internal memory on each data phase (beat). Therefore, a read from the internal memory occurs only when the read data phase (beat) actually starts on the protocol signals. For a write, data is written to the internal memory as soon as each individual write data phase (beat) completes.

Note

In addition to the above functions, you can configure other aspects of the AXI3 Slave BFM by using the functions: "set_config()" on page 148 and "get_config()" on page 149.

Using the Basic Slave API

As described in the *VHDL Basic Slave API Definition* section, there are a set of procedures that you use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing. However, consider the following configurations when using the Slave Test program.

- *slave_mode* - the read and write channel interaction may cause simultaneous read and write transactions to occur at the same address. With the default *slave_mode* setting the read transaction, data burst is buffered at the start of the burst and the write data burst is buffered at the end of the burst. This can result in the read data being stale at the time it is transmitted over the protocol signals. If this is an undesirable result, then set the *slave_mode* to be *AXI_PHASE_SLAVE*.
- *delay_mode* - by default, the handshake **READY* signal always follows, or is simultaneous with the **VALID* signal. By configuring the *delay_mode* to be *AXI_TRANS2READY, *READY* before **VALID* scenarios can be achieved.

VHDL Advanced Slave API Definition

The remaining section of this tutorial presents a walk-through of the Advanced Slave API. It consists of three processes for write transactions and two for read transactions. You are not required to edit the following Advance Slave API, unless you require a different response than the default (*OKAY*) response.

The *process write* code extract demonstrates how to create and process multiple write transactions. Each write transaction has a unique index number associated with it that increments for each new transaction. A *write_trans* variable is defined to hold this transaction index.

The processing of write transactions begins after the system reset becomes inactive. A loop then creates a slave transaction for the BFM indexed by the *index* argument. Within the loop, the delay for the *AWREADY* signal is set by the *set_write_address_ready_delay* procedure and then it waits for a write address phase to occur using the *get_write_addr_phase* procedure. The transaction record is then pushed onto a transaction queue with its unique *write_trans* transaction index and queue identifier *AXI_QUEUE_ID_0* by the *push_transaction_id* procedure.

The loop then completes and starts again by creating a new transaction and waiting for another write address phase to occur (refer to Example 11-13).

Example 11-13. process write

```
-- process_write : write address phase through path 0
-- This process keep receiving write address phases and pushes
-- the transaction into a queue via the push_transaction_id procedure.
process
    variable write_trans : integer;
begin
    wait_on(AXI_RESET_POSEDGE, index, axi_tr_if_0(index));
    loop
      create_slave_transaction(write_trans, index, axi_tr_if_0(index));
      set_write_address_ready_delay(write_trans, axi_tr_if_0(index));
      get_write_addr_phase(write_trans, index, axi_tr_if_0(index));
      push_transaction_id(write_trans, AXI_QUEUE_ID_0, index,
axi_tr_if_0(index));
    end loop;
    wait;
end process;
```

The *handle write* code extract demonstrates how to write a data burst to an internal memory using buffered and unbuffered approaches.

To perform pipelining of the AXI3 address and data phases of one transaction concurrently with another transaction, the Advanced Slave API provides procedures with an optional *path_id* argument. This permits multiple threads of code execution within the Slave API. The following handle write code uses *path_id* = `AXI_PATH_1`, as an example.

Initially, a number of local variables are defined to hold the transaction index *write_trans* and some of the transaction fields, such as burst length, data, etc. Within a loop, the existing record of the *write_trans* transaction is popped from the queue identifier *AXI_QUEUE_ID_0* via the *pop_transaction_id* procedure. The delay for the *WREADY* signal is then set by the *set_write_data_ready_delay* procedure.

If the *slave_mode* is configured to *AXI_TRANSACTION_SLAVE* (buffered) the code waits for a complete write data burst by the *get_write_data_burst* procedure before continuing. The burst length of the write data burst is obtained using the *get_burst_length* procedure. The resulting *burst_length* is then used to set the subsequent maximum inner loop *i* count for the number of data beats within the burst. Loop *i* gets the address and data pairs from the transaction via the

get_write_addr_data procedure before calling the *do_byte_write* procedure which writes the *data* byte into the memory *mem* at the corresponding *addr* address. If the number of bytes to be written (for this beat) is more than one, then loop *j* writes the remaining bytes of this beat into the memory *mem*. Loop *i* then repeats for each data beat up to the length of the data burst.

If the slave_mode is configured to *AXI_PHASE_SLAVE* (unbuffered) the code waits for a single write data phase (beat) to complete via the *get_write_data_phase* procedure, within a *while* loop. The address and data pairs from the transaction are obtained by the *get_write_addr_data* procedure before calling the *do_byte_write* procedure which writes the *data* byte into the memory *mem* at the corresponding *addr* address. If the number of bytes to be written (for this beat) is more than one, then loop *j* writes the remaining bytes of this beat into the memory *mem*. The *while* loop then repeats, waiting for another data phase (beat), if this is not the last data phase (beat) in the burst.

The transaction record, so far, is then pushed onto a new transaction queue identifier *AXI_QUEUE_ID_2* by the *push_transaction_id* procedure, which is then ready for the slave to execute a response back to the master (refer to Example 11-14).

Example 11-14. handle write

```
-- handle_write : write data phase through path 1
-- This process receives write data burst, or write phases (beats).
-- The slave_mode configuration controls when the write data is passed to
-- memory.
  process
    variable write_trans: integer;
    variable byte_length : integer;
    variable burst_length : integer;
    variable addr : std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
    variable data : std_logic_vector(7 downto 0);
    variable last : integer := 0;
    variable loop_i : integer := 0;
  begin
    loop
      pop_transaction_id(write_trans, AXI_QUEUE_ID_0, index, AXI_PATH_1,
axi_tr_if_1(index));
      set_write_data_ready_delay(write_trans, AXI_PATH_1,
axi_tr_if_1(index));

      if (slave_mode = AXI_TRANSACTION_SLAVE) then
        get_write_data_burst(write_trans, index, AXI_PATH_1,
axi_tr_if_1(index));
        get_burst_length(burst_length, write_trans, index, AXI_PATH_1,
axi_tr_if_1(index));
        for i in  0 to burst_length loop
         get_write_addr_data(write_trans,  i, 0, byte_length, addr, data,
index, AXI_PATH_1, axi_tr_if_1(index));
          do_byte_write(addr, data);
          if byte_length > 1 then
            for j in 1 to byte_length-1 loop
               get_write_addr_data(write_trans,  i, j, byte_length, addr,
data, index, AXI_PATH_1, axi_tr_if_1(index));
              do_byte_write(addr, data);
```

```
            end loop;
          end if;
        end loop;
      else
        last := 0;
        loop_i := 0;
        while(last = 0) loop
          get_write_data_phase(write_trans, loop_i, last, index,
AXI_PATH_1, axi_tr_if_1(index));
          loop_i := loop_i + 1;
         get_write_addr_data(write_trans,  loop_i, 0, byte_length, addr,
data, index, AXI_PATH_1, axi_tr_if_1(index));
          do_byte_write(addr, data);
          if byte_length > 1 then
            for j in 1 to byte_length-1 loop
               get_write_addr_data(write_trans,  loop_i, j, byte_length,
addr, data, index, AXI_PATH_1, axi_tr_if_1(index));
               do_byte_write(addr, data);
            end loop;
          end if;
        end loop;
      end if;
      push_transaction_id(write_trans, AXI_QUEUE_ID_2, index, AXI_PATH_1,
axi_tr_if_1(index));
    end loop;
    wait;
  end process;
```

The *handle response* code extract demonstrates how to respond to a master write transaction using a different *path_id = AXI_PATH_2* than the path_id used for the address and data phases of the same transaction. This gives the ability for the slave to execute a write transaction response in a different order than the order received for a particular *write_trans* index number.

A *write_trans* variable is defined to hold this transaction index number before entering a loop to pop a write transaction from *AXI_QUEUE_ID_2* via the *pop_transaction_id* procedure call. The delay for the *BVALID* signal is then set via the *set_wr_resp_valid_delay* procedure. The response phase is then executed by the *execute_write_response_phase* procedure call (refer to Example 11-15).

Example 11-15. handle response

```
-- handle_response : write response phase through path 2
-- This method sends the write response phase
process
    variable write_trans: integer;
  begin
    loop
      pop_transaction_id(write_trans, AXI_QUEUE_ID_2, index, AXI_PATH_2,
axi_tr_if_2(index));
      set_wr_resp_valid_delay(write_trans, AXI_PATH_2,
axi_tr_if_2(index));
      execute_write_response_phase(write_trans, index, AXI_PATH_2,
axi_tr_if_2(index));
```

```
    end loop;
    wait;
  end process;
```

The processing of read transactions works in a similar way as that described above for write transactions. There are two processes *process_read* and *handle read*.

The main difference between write and read transaction handling is that the read transaction retrieves the read data burst from the internal memory *mem* at the start of the data burst, or on a phase-by-phase (beat-by-beat) basis, depending on the *slave_mode* configuration setting. In addition AXI3 has a read response per read data phase (beat), so unlike the write, read does not require a separate read response handling process (refer to Example 11-16 and Example 11-17).

Example 11-16. process_read

```
-- process_read : read address phase through path 3
  -- This process keep receiving read address phase and push the
transaction into queue through
  -- push_transaction_id API.
  process
    variable read_trans: integer;
  begin
    wait_on(AXI_RESET_POSEDGE, index, AXI_PATH_3, axi_tr_if_3(index));
    loop
      create_slave_transaction(read_trans, index, AXI_PATH_3,
axi_tr_if_3(index));
      set_read_address_ready_delay(read_trans, AXI_PATH_3,
axi_tr_if_3(index));
      get_read_addr_phase(read_trans, index, AXI_PATH_3,
axi_tr_if_3(index));
      push_transaction_id(read_trans, AXI_QUEUE_ID_1, index, AXI_PATH_3,
axi_tr_if_3(index));
    end loop;
    wait;
  end process;
```

Example 11-17. handle read

```
-- handle_read : read data and response through path 4
  -- This process reads data from memory and send read data/response
either at
  -- burst or phase level depending upon slave working mode.
  process
    variable read_trans: integer;
    variable burst_length : integer;
    variable byte_length : integer;
    variable addr : std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
    variable data : std_logic_vector(7 downto 0);
  begin
    loop
      pop_transaction_id(read_trans, AXI_QUEUE_ID_1, index, AXI_PATH_4,
axi_tr_if_4(index));
      set_read_data_valid_delay(read_trans, AXI_PATH_4,
axi_tr_if_4(index));
```

```
      get_burst_length(burst_length, read_trans, index, AXI_PATH_4,
axi_tr_if_4(index));
      for i in  0 to burst_length loop
        get_read_addr(read_trans,  i, 0, byte_length, addr, index,
AXI_PATH_4, axi_tr_if_4(index));
        do_byte_read(addr, data);
        set_read_data(read_trans, i, 0, byte_length, addr, data, index,
AXI_PATH_4, axi_tr_if_4(index));
        if slave_mode = AXI_PHASE_SLAVE then
          execute_read_data_phase(read_trans, i, index, AXI_PATH_4,
axi_tr_if_4(index));
        end if;
        if byte_length > 1 then
          for j in 1 to byte_length-1 loop
             get_read_addr(read_trans,  i, j, byte_length, addr, index,
AXI_PATH_4, axi_tr_if_4(index));
             do_byte_read(addr, data);
             set_read_data(read_trans, i, j, byte_length, addr, data,
index, AXI_PATH_4, axi_tr_if_4(index));
             if slave_mode = AXI_PHASE_SLAVE then
               execute_read_data_phase(read_trans, i, index, AXI_PATH_4,
axi_tr_if_4(index));
             end if;
          end loop;
        end if;
      end loop;
      if slave_mode = AXI_TRANSACTION_SLAVE then
        execute_read_data_burst(read_trans, index, AXI_PATH_4,
axi_tr_if_4(index));
      end if;
    end loop;
    wait;
  end process;
```

Chapter 12
Getting Started with Qsys and the BFMs

This example shows you how to use the Qsys tool in the Quartus II software to create a top-level design environment. You will be using the *ex1_back_to_back_sv* example, a SystemVerilog example from the *$QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi3/qsys-examples* directory in the Altera Complete Design Suite (ACDS) installation.

You will do the following tasks:

1. Create a work directory
2. Copy the example to the work directory
3. Invoke Qsys from the Quartus II sotware Tools menu
4. Generate a top-level netlist
5. Run simulation by referencing the README and scripts for your simulation environment

Setting Up the Simulation Environment

The following steps outline how to do the first four steps above and use Qsys to set up the simulation environment.

1. Create a work directory into which you copy the example file, *ex1_back_to_back_sv* from the Installation.

   a. Create a work directory into which you will copy the example file.

   ```
   mkdir axi3-qsys-example
   ```

   b. Copy the example file from the installation directory into your work directory.

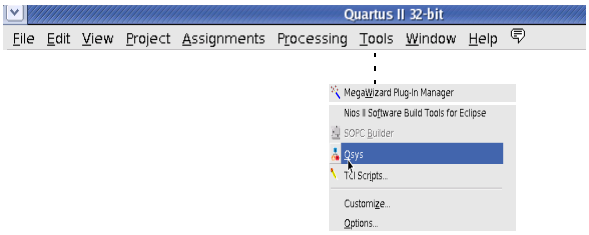
   ```
   cp -r $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi3/\
      qsys-examples/* axi3-qsys-example/
   ```

2. Change the directory path to your local path where the example resides:

   ```
   cd axi3-qsys-example/ex1_back_to_back_sv
   ```

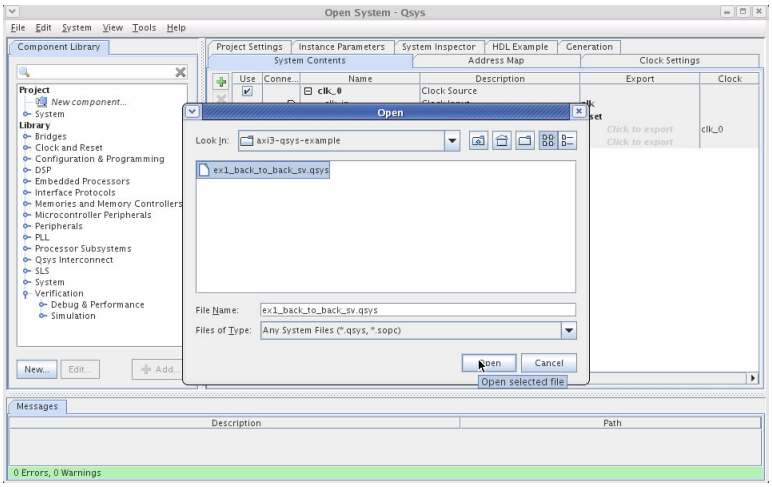
3. Open Qsys in the Quartus II software menu.
   Start the Quartus II software. When the Quartus II GUI appears, select *Tools>Qsys* (refer to Figure 12-1).

Figure 12-1. Select Qsys from the Quartus II Software Top-Level Menu



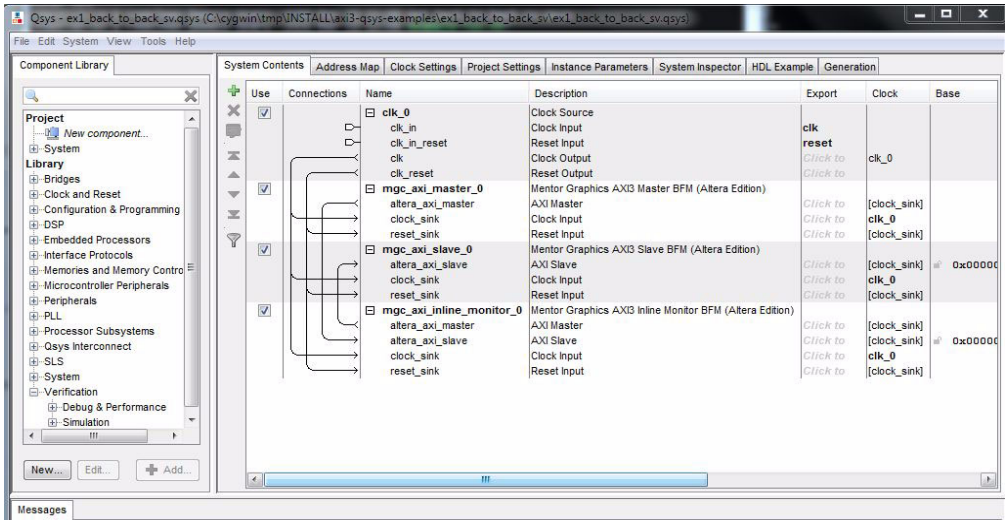
4. Qsys opens and displays a dialog box from which you select a file with the *qsys* filetype. Select the file *ex1_back_to_back_sv.qsys* and click *Open* (refer to Figure 12-2).

Figure 12-2. Select and Open the Example File



5. Qsys displays the connectivity of the selected example as shown in Figure 12-3.

Figure 12-3. Quartus II Software Displays Connectivity of the Example

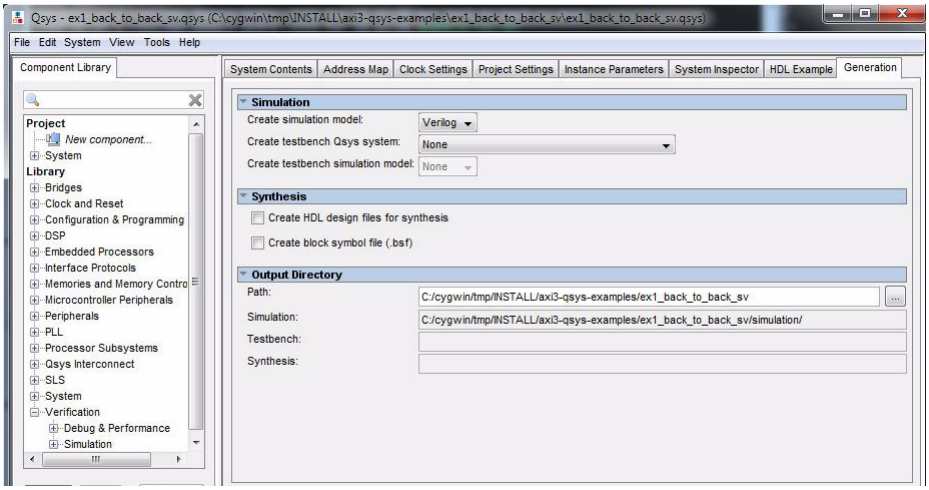


Note

If you are using VHDL, you must select each BFM and verify that the index number specified for the BFM is correct. An information dialog displays the properties of the BFM when you select it. Ensure the specified BFM *index* is correct in this dialog. If you do not know the correct index number, check the VHDL code for the BFM.

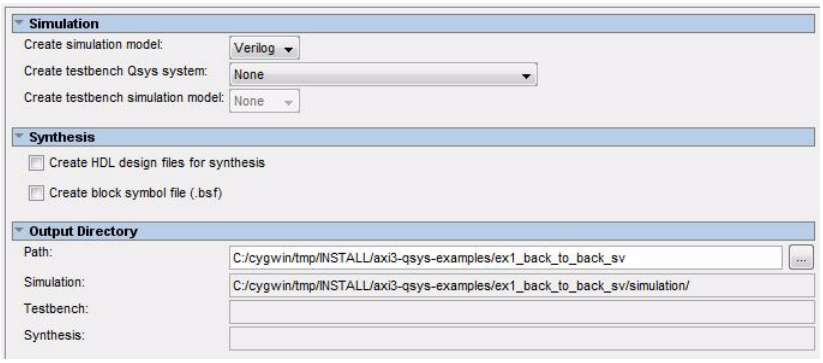
6. Select the *Generation* tab on the upper right side of the window (refer to Figure 12-3) to display the netlist generation options.
7. Specify the netlist generation options and generate the netlist (Figure 12-4).

Figure 12-4. Generation Tab Window



8. Now change the example's path, select the simulation model, and turn off synthesis as outlined in the steps below and shown in Figure 12-5.

Figure 12-5. Set Path, Simulation and Synthesis Options



a. Change the example's path. In the Path field of the Output Directory section, ensure the path correctly specifies the subdirectory *ex1_back_to_back_sv*, which is the subdirectory containing the example that you just copied into a temporary directory.

Note

If this example's subdirectory name is duplicated in the example's path specification, you must remove one of the duplicated subdirectory names.

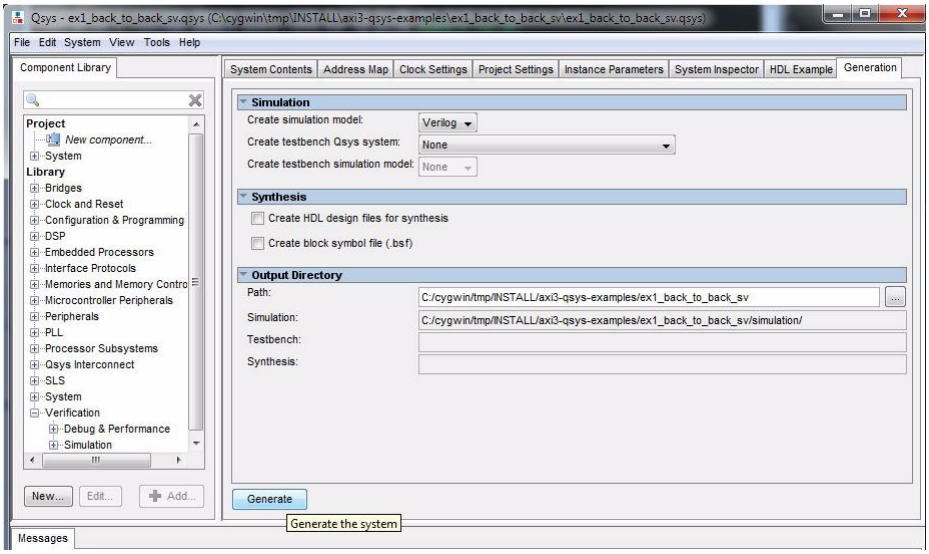
The example's path specified in the Path field of the Output Directory section **must be correct before** selecting *Verilog* or *VHDL* in the next step.

b. Under the Simulation section, select the language for your output files, *Verilog* or *VHDL* , for the value of the *Create simulation model* option (refer to Figure 12-5). The Qsys testbench options should be unchecked.

c. Uncheck the options under the *Synthesis* section to turn off generation of files for synthesis (refer to Figure 12-5).

d. Click the *Generate* button on the bottom left side of the window (refer to Figure 12-6).

Figure 12-6. Click Generate



e. Now, refer to the section Running Simulation for steps to start simulation.

Running Simulation

You can run simulation either from a GUI interface or a command line. However, before starting simulation, you must define the following:

- Check to ensure the $QUARTUS_ROOTDIR environment variable points to the Quartus II software directory in the Quartus II software installation. The example script (*example.do* or *example.vhld.do*) uses this variable to locate the Mentor VIP AE BFM during simulation.
- When using Mentor VIP AE BFM, you must ensure when you start simulation that the variable *MvcHome* points to the location of the installed Mentor VIP AE BFM. You can set the location of *MvcHome* using one of the following ways:
  - Set the *MvcHome* variable in the *modelsim.ini* file, refer to the section "Using a Shortcut or Editing the modelsim.ini File".
  - Specify the *-mvchome* option on the command line as shown in the command line examples in the section "Invoking Simulation From a Command Line".

The command and script you specify to run simulation varies based on the simulator. Table 12-1 outlines the file and script names you specify in your simulator command.

Table 12-1. Simulator and Script Names

	Questa Simulation	ModelSim Simulation	IUS Simulation	VCS Simulation
README	README-Questa.txt	README-ModelSim.txt	README-IUS.txt	README-VCS.txt
Script Name	example.do	example.do	example-ius.sh	example-vcs.sh

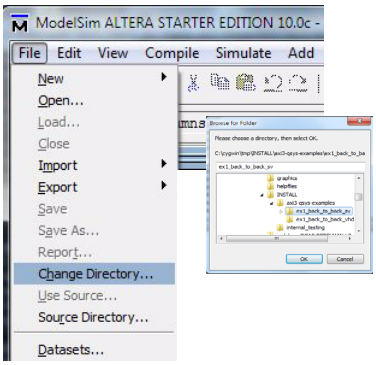
Using the Modelsim simulator, the following sections outline how to run simulation from either a GUI or command line.

Invoking Simulation From a GUI

Using the ModelSim simulator, this section outlines how to invoke a simulator from a GUI.

1. Start the ModelSim GUI.
2. Select the *File* menu and then click the *Change Directory* option. When the Browse for Folder dialog appears, select the work area directory that contains the example.

Figure 12-7. Select the Work Directory



3. Start the example script from the Transcript window by typing the following commands:

```
ModelSim> do example.do
ModelSim> run -all
```

The first command above starts the script that compiles and elaborates the test programs. The second ModelSim command starts simulation and runs the simulation until simulation completes.

For details about the processing done by the example script, refer to the section "Example Script Processing".

Invoking Simulation From a Command Line

This section outlines the commands that you use to run simulation from a command line specified from either a UNIX shell or a Windows command window.

1. If you are using ModelSim for simulation, set the shared library path with the *mvchome* option.

```
modelsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
```

2. Change the directory to the work area containing the example.

```
cd axi3-qsys-examples/ex1_back_to_back_sv
```

3. At the ModelSim prompt, start the example script.

```
ModelSim> do example.do
```

Alternatively, you can launch the simulator and run the script with one command. However, you must first set the directory to the work area containing the example as the following commands illustrate.

1. Change the directory to the work area containing the example.

```
cd axi3-qsys-examples/ex1_back_to_back_sv
```

2. Starting from a UNIX or Windows process, invoke *Modelsim* on the *example.do* script by typing the following command.

```
modelsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/\
      mentor_vip_ae/common -gui -do example.do
```

3. When ModelSim starts, type the run -all command:

```
ModelSim> run -all
```

Note

If your example is a VHDL example, the script name is *example_vhdl.do*

For details on the processing done by the script, refer to Example Script Processing.

Example Script Processing

After starting one of the example scripts from Table 12-1, the script compiles the Mentor VIP AE BFM for AXI3, then invokes two *tcl* aliases—*dev_com* and *com* to compile the required design files. These alias commands are defined in the *msim_setup.tcl* simulation script generated by Qsys along with the simulation model files.

Next, *vlog* compiles the three test programs:

- *master_test_program.sv*
- *slave_test_program.sv*
- *monitor_test_program.sv*

The script then uses *vlog* again to compile the *top.sv*. Simulation starts with the *elab* alias.

```
set TOP_LEVEL_NAME top
set QSYS_SIMDIR    simulation

source $QSYS_SIMDIR/mentor/msim_setup.tcl
if {![info exists env(MENTOR_VIP_AE)]} {
  set env(MENTOR_VIP_AE)
$env(QUARTUS_ROOTDIR)/../ip/altera/mentor_vip_ae
}

  ensure_lib libraries
  ensure_lib libraries/work
```

```
  vmap work  libraries/work

  vlog -work work -sv \
    $env(MENTOR_VIP_AE)/common/questa_mvc_svapi.svh \
    $env(MENTOR_VIP_AE)/axi3/bfm/mgc_common_axi.sv \
    $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_monitor.sv \
    $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_inline_monitor.sv \
    $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_master.sv \
    $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_slave.sv

# Compile device library files
dev_com

# Compile Qsys-generated design files
com

# Compile example test program files
vlog  master_test_program.sv
vlog   slave_test_program.sv
vlog monitor_test_program.sv

# Compile top-level design file
vlog top.sv

# Simulate
elab
```

Using a Shortcut or Editing the modelsim.ini File

On Windows, if you use a shortcut or if you want to ensure the path specification is done before simulation, edit the *modelsim.ini* file, which will provide the path to the Mentor VIP AE installation for all future invocations of ModelSim or Questa.

To edit the ModelSim/Questa installation directory follow these steps:

1. Editing the file *modelsim.ini*, find the section that starts with *[vsim]*.
2. Search for *MvcHome*.
3. Add or change the *MvcHome* path to point to the location where the Mentor VIP AE is installed. Do not forget the *common* subdirectory. For example:

```
MvcHome = $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
```

Note

Setting *MvcHome* to the *modelsim.ini* file eliminates the need to specify the *-mvchome* option on the vsim command line.

Do not use *vmap* to specify the installed location of Mentor VIP AE. Using *vmap* puts the *MvcHome* in the [library] section of *modelsim.ini*.

```
vmap MvcHome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
```

Note

ModelSim Altera Edition and ModelSim Altera Starter Edition do not have a default path defined for the variable *MvcHome*. If you are using either of these simulators, you must define a path for this variable by adding a new entry to the file. Be sure to add the entry in the *[vsim]* section of the file (after the notation *[vsim]*, and before the next section).

Appendix A
SystemVerilog
Master and Slave Test Programs

SystemVerilog Master Test Program

The example code in this section is a simplified AXI master which illustrates how you can use the *mgc_axi_master* BFM.

```
        //
        ******************************************************************
        //
        // Copyright 2007-2012 Mentor Graphics Corporation
        // All Rights Reserved.
        //
        // THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH
        // IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
        // AND IS SUBJECT TO LICENSE TERMS.
        //
        ******************************************************************

        /*
            This simplified example of an AXI master shows you how to use the
        mgc_axi_master BFM. This master performs a directed test, initiating
        four sequential writes, followed by four sequential reads. It then
        verifies that the data read out matches the data written.
            For simplicity, only one data cycle is used (default AXI burst
        length encoding 0). It then initiates two write data bursts,
        followed by two read data bursts.
        */

        import mgc_axi_pkg::*;
        module master_test_program #(int AXI_ADDRESS_WIDTH = 32, int
        AXI_RDATA_WIDTH = 1024, int AXI_WDATA_WIDTH = 1024, int AXI_ID_WIDTH
        = 4)
        (
    mgc_axi_master bfm
);

initial
begin
    axi_transaction trans;
    /******************
    ** Configuration **
    ******************/
    begin
      bfm.set_config(AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000);

end
```

```
    /*******************
    ** Initialization **
    *******************/
    bfm.wait_on(AXI_RESET_POSEDGE);
    bfm.wait_on(AXI_CLOCK_POSEDGE);

    /************************
    ** Traffic Generation: **
    ************************/
    // 4 x Writes
    // Write data value 1 on byte lanes 1 to address 1.
    trans = bfm.create_write_transaction(1);
    trans.set_data_words(32'h0000_0100, 0);
    trans.set_write_strobes(4'b0010, 0);
    $display ( "@ %t, master_test_program: Writing data (1) to address
(1)", $time);

    // By default it will run in Blocking mode
    bfm.execute_transaction(trans);

    // Write data value 2 on byte lane 2 to address 2.
    trans = bfm.create_write_transaction(2);
    trans.set_data_words(32'h0002_0000, 0);
    trans.set_write_strobes(4'b0100, 0);
    trans.set_write_data_mode(AXI_DATA_WITH_ADDRESS);
    $display ( "@ %t, master_test_program: Writing data (2) to address
(2)", $time);

    bfm.execute_transaction(trans);

    // Write data value 3 on byte lane 3 to address 3.
    trans = bfm.create_write_transaction(3);
    trans.set_data_words(32'h0300_0000, 0);
    trans.set_write_strobes(4'b1000, 0);
    $display ( "@ %t, master_test_program: Writing data (3) to address
(3)", $time);

    bfm.execute_transaction(trans);

    // Write data value 4 to address 4 on byte lane 0.
    trans = bfm.create_write_transaction(4);
    trans.set_data_words(32'h0000_0004, 0);
    trans.set_write_strobes(4'b0001, 0);
    trans.set_write_data_mode(AXI_DATA_WITH_ADDRESS);
    $display ( "@ %t, master_test_program: Writing data (4) to address
(4)", $time);

    bfm.execute_transaction(trans);

    // 4 x Reads
    // Read data from address 1.
    trans = bfm.create_read_transaction(1);
    trans.set_id(1);
```

```
    bfm.execute_transaction(trans);
    if (trans.get_data_words(0) == 32'h0000_0100)
        $display ( "@ %t, master_test_program: Read correct data (1) at
address (1)", $time);
    else
       $display ( "@ %t master_test_program: Expected data (1) at address
1, but got %d", $time, trans.get_data_words(0));

    // Read data from address 2.
    trans = bfm.create_read_transaction(2);
    trans.set_id(2);

    bfm.execute_transaction(trans);
    if (trans.get_data_words(0) == 32'h0002_0000)
        $display ( "@ %t, master_test_program: Read correct data (2) at
address (2)", $time);
    else
       $display ( "@ %t, master_test_program: Expected data (2) at address
2, but got %d", $time, trans.get_data_words(0));

    // Read data from address 3.
    trans = bfm.create_read_transaction(3);
    trans.set_id(3);

    bfm.execute_transaction(trans);
    if (trans.get_data_words(0) == 32'h0300_0000)
      $display ( "@ %t, master_test_program: Read correct data (3) at
address (3)", $time);
    else
       $display ( "@ %t, master_test_program: Expected data (3) at address
3, but got %d", $time, trans.get_data_words(0));

    // Read data from address 4.
    trans = bfm.create_read_transaction(4);
    trans.set_id(4);

    bfm.execute_transaction(trans);
    if (trans.get_data_words(0) == 32'h0000_0004)
        $display ( "@ %t, master_test_program: Read correct data (4) at
address (4)", $time);
    else
       $display ( "@ %t, master_test_program: Expected data (4) at address
4, but got %d", $time, trans.get_data_words(0));

    // Write data burst length of 7 to start address 16.
    trans = bfm.create_write_transaction(16, 7);

    trans.set_size(AXI_BYTES_4);
    trans.set_data_words(`hACE0ACE1, 0);
    trans.set_data_words(`hACE2ACE3, 1);
    trans.set_data_words(`hACE4ACE5, 2);
    trans.set_data_words(`hACE6ACE7, 3);
    trans.set_data_words(`hACE8ACE9, 4);
    trans.set_data_words(`hACEAACEB, 5);
    trans.set_data_words(`hACECACED, 6);
    trans.set_data_words(`hACEEACEF, 7);
    for(int i=0; i<8; i++)
              trans.set_write_strobes(4'b1111, i);
```

```
    trans.set_write_data_mode(AXI_DATA_WITH_ADDRESS);
    $display ( "@ %t, master_test_program: Writing data burst of length 7
to start address 16", $time);

    bfm.execute_transaction(trans);

    // Write data burst of length 7 to start address 128 with LSB write
    // strobe inactive.
    trans = bfm.create_write_transaction(128, 7);

    trans.set_size(AXI_BYTES_4);
    trans.set_data_words(`hACE0ACE1, 0);
    trans.set_data_words(`hACE2ACE3, 1);
    trans.set_data_words(`hACE4ACE5, 2);
    trans.set_data_words(`hACE6ACE7, 3);
    trans.set_data_words(`hACE8ACE9, 4);
    trans.set_data_words(`hACEAACEB, 5);
    trans.set_data_words(`hACECACED, 6);
    trans.set_data_words(`hACEEACEF, 7);

    trans.set_write_strobes(4'b1110, 0);
    for(int i=1; i<8; i++)
        trans.set_write_strobes(4'b1111, i);
    $display ( "@ %t, master_test_program: Writing data burst of length 7
to start address 128", $time);

    bfm.execute_transaction(trans);

    // Read data burst of length 1 from address 16.
    trans = bfm.create_read_transaction(16, 1);
    trans.set_size(AXI_BYTES_4);

    bfm.execute_transaction(trans);
    if (trans.get_data_words(0) == `hACE0ACE1)
      $display ( "@ %t, master_test_program: Read correct data (hACE0ACE1)
at address (16)", $time);
    else
      $display ( "@ %t, master_test_program: Expected data (hACE0ACE1) at
address (16), but got %h", $time, trans.get_data_words(0));

    if (trans.get_data_words(1) == `hACE2ACE3)
      $display ( "@ %t, master_test_program: Read correct data (hACE2ACE3)
at address (20)", $time);
    else
      $display ( "@ %t, master_test_program: Expected data (hACE2ACE3) at
address (20), but got %h", $time, trans.get_data_words(1));


    // Read data burst of length 1 from address 128.
    trans = bfm.create_read_transaction(128, 1);
    trans.set_size(AXI_BYTES_4);
```

```
    bfm.execute_transaction(trans);
    if (trans.get_data_words(0) == 'hACE0AC00)
      $display ( "@ %t, master_test_program: Read correct data (hACE0AC00)
at address (128)", $time);
    else
      $display ( "@ %t, master_test_program: Expected data (hACE0AC00) at
address (128), but got %h", $time, trans.get_data_words(0));

    if (trans.get_data_words(1) == 'hACE2ACE3)
      $display ( "@ %t, master_test_program: Read correct data (hACE2ACE3)
at address (132)", $time);
    else
      $display ( "@ %t, master_test_program: Expected data (hACE2ACE3) at
address (132), but got %h", $time, trans.get_data_words(1));

    #100
    $finish();
end
endmodule
```

SystemVerilog Slave Test Program

This section contains a simple example of an AXI slave, illustrating how to use the *mgc_axi_slave* BFM.

```
//
********************************************************************************
//
// Copyright 2007-2012 Mentor Graphics Corporation
// All Rights Reserved.
//
// THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS
THE PROPERTY OF
// MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE
TERMS.
//
//
********************************************************************************
This simple example of an AXI Slave demonstrates the mgc_axi_slave BFM
usage. This slave BFM can handle most write and read transaction scenarios
from the master, including a write data burst starting at the same time,
or after, its address phase.

/*
    This slave code is divided into two parts, one which the user edits to
change slave mode (Transaction/burst or Phase level)and memory handling.
    Code labeled in this example that the "Users normally need not edit
this code", sometimes requires the user to edit it, if the code must meet
required phase valid/ready delays.
*/
```

```
import mgc_axi_pkg::*;

module slave_test_program #(int AXI_ADDRESS_WIDTH = 32, int
AXI_RDATA_WIDTH = 32, int AXI_WDATA_WIDTH = 32, int AXI_ID_WIDTH = 4)
(
    mgc_axi_slave bfm
);

  typedef bit [((AXI_ADDRESS_WIDTH) - 1) : 0] addr_t;

  // Enum type for slave mode
  // AXI_TRANSACTION_SLAVE - Works at burst level (write data is received
at
  //                         burst and read data/response is sent in burst)
  // AXI_PHASE_SLAVE       - Write data and read data/response is worked
upon
  //                          at phase level
  typedef enum bit
  {
    AXI_TRANSACTION_SLAVE = 1'b0,
    AXI_PHASE_SLAVE       = 1'b1
  } axi_slave_mode_e;

  ////////////////////////////////////////////////
  // Users can edit the following code to meet requirements.
// For example:
// - Modify the default slave_mode configuration from
// AXI_TRANSACTION_SLAVE (buffered) to AXI_PHASE_SLAVE (unbuffered)
// for read/write data bursts.
// - Modify the default memory to a FIFO for FIXED transactions
// - Modify the delays of the *VALID and *READY signals to create
// a *READY active before *VALID active scenario
  ////////////////////////////////////////////////

  // Slave mode selection : default it is transaction level slave
  axi_slave_mode_e slave_mode = AXI_TRANSACTION_SLAVE;

  // Storage for a memory
  bit [7:0] mem [*];

  // Function: do_byte_read
  // Function provides read data byte from memory at particular input
  // address
  function bit[7:0] do_byte_read(addr_t addr);
    return mem[addr];
  endfunction

  // Function: do_byte_write
  // Function writes data byte to memory at particular input address
  function void do_byte_write(addr_t addr, bit [7:0] data);
    mem[addr] = data;
  endfunction

  // Function: set_write_address_ready_delay
  // This function sets and extends the write address phase ready delay
  function void set_write_address_ready_delay(axi_transaction trans);
    trans.set_address_ready_delay(2);
  endfunction
```

```
  // Function: set_read_address_ready_delay
  // This sets read address phase ready delay to extend the phase
  function void set_read_address_ready_delay(axi_transaction trans);
    trans.set_address_ready_delay(4);
  endfunction

  // Function: set_write_data_ready_delay
  // This function sets the ready delays for each write data phase in a
  //write data burst.
  function void set_write_data_ready_delay(axi_transaction trans);
    for (int i = 0; i < trans.data_ready_delay.size(); i++)
      trans.set_data_ready_delay(i, i);
  endfunction

  // Function: set_wr_resp_valid_delay
  // This sets write response phase valid delay to start driving the
  // write response phase after a specified delay.
  function void set_wr_resp_valid_delay(axi_transaction trans);
    trans.set_write_response_valid_delay(2);
  endfunction

  // Function: set_read_data_valid_delay
  // This sets read response phase valid delays to start driving the
  // read data/response phases after a specified delay.
  function void set_read_data_valid_delay(axi_transaction trans);
    for (int i = 0; i < trans.data_valid_delay.size(); i++)
      trans.set_data_valid_delay(i, i);
  endfunction


  ////////////////////////////////////////////////////////////////////////
  // Users normally need not edit this code.
  // The following code constitutes the Advanced Slave API.
 // You do not need to edit the following code unless you need to change
 // the AXI3 default response of OKAY
 //
// This code also assigns the handshake signal delays for *VALID and
// *READY. You may want to change the default delays according to
// your own requirements. For example, to create a *READY active before
// *VALID active scenario.
//
  // address_ready_delay : This is for both write and read address phase
  // data_valid_delay    : This is for sending read data/response valid
  // data_ready_delay    : This is for write data phase ready delay
  ////////////////////////////////////////////////////////////////////////
  initial
  begin
    // Initialisation
    wait (bfm.ARESETn == 1);

    // Traffic generation
    fork
      process_read;
      process_write;
    join
  end
```

```
// Task: process_read
// This method continues to receive read address phase and calls
// another method to process the received transaction.
task process_read;
  forever
  begin
    axi_transaction read_trans;

    read_trans = bfm.create_slave_transaction();
    set_read_address_ready_delay(read_trans);
    bfm.get_read_addr_phase(read_trans);

    fork
      begin
        automatic axi_transaction t = read_trans;
        handle_read(t);
      end
    join_none
    #0;
  end
endtask

// Task: handle_read
// This method reads data from memory and sends the read data/response
// either at burst or phase level, based on the slave mode.
task automatic handle_read(input axi_transaction read_trans);
  addr_t addr[];
  bit [7:0] mem_data[];

  set_read_data_valid_delay(read_trans);

  for(int i = 0; bfm.get_read_addr(read_trans, i, addr); i++)
  begin
    mem_data = new[addr.size()];
    for (int j = 0; j < addr.size(); j++)
      mem_data[j] = do_byte_read(addr[j]);

    bfm.set_read_data(read_trans, i, addr, mem_data);

    if (slave_mode == AXI_PHASE_SLAVE)
      bfm.execute_read_data_phase(read_trans, i);
  end

  if (slave_mode == AXI_TRANSACTION_SLAVE)
    bfm.execute_read_data_burst(read_trans);
endtask

// Task: process_write
// This method continues to receive write address phase and calls
// another method to process the received transaction.
task process_write;
  forever
  begin
    axi_transaction write_trans;
```

```
      write_trans = bfm.create_slave_transaction();
      set_write_address_ready_delay(write_trans);
      bfm.get_write_addr_phase(write_trans);

      fork
        begin
          automatic axi_transaction t = write_trans;
          handle_write(t);
        end
      join_none
      #0;
    end
  endtask

  // Task: handle_write
  // This method receives write data burst or phases for a write
  // transaction based on the slave mode, write data to memory, and then
  // sends a response.
  task automatic handle_write(input axi_transaction write_trans);
    addr_t addr[];
    bit [7:0] data[];
    bit last;

    set_write_data_ready_delay(write_trans);

    if (slave_mode == AXI_TRANSACTION_SLAVE)
    begin
      bfm.get_write_data_burst(write_trans);

     for( int i = 0; bfm.get_write_addr_data(write_trans, i, addr, data);
i++ )
      begin
        for (int j = 0; j < addr.size(); j++)
          do_byte_write(addr[j], data[j]);
      end
    end
    else
    begin
      for(int i = 0; (last == 1'b0); i++)
      begin
        bfm.get_write_data_phase(write_trans, i, last);

        void'(bfm.get_write_addr_data(write_trans, i, addr, data));
        for (int j = 0; j < addr.size(); j++)
          do_byte_write(addr[j], data[j]);
      end
    end

    set_wr_resp_valid_delay(write_trans);
    bfm.execute_write_response_phase(write_trans);
  endtask

endmodule
```

Appendix B
VHDL Master and Slave Test Programs

This appendix contains two VHDL code examples, one for the Master BFM and the other for the Slave BFM.

VHDL Master BFM Code Example

The following code example shows you a simplified AXI Master, which demonstrates how to use the Master.

```
******************************************************************************
--
-- Copyright 2007-2012 Mentor Graphics Corporation
-- All Rights Reserved.
--
-- THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS
-- THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS
-- SUBJECT TO LICENSE TERMS.
--
******************************************************************************

-- This simplified example of an AXI master shows you how to use the
-- mgc_axi_master BFM.
--
-- This master performs a directed test, initiating 4 sequential writes,
-- followed by 4 sequential reads. It then verifies that the data read out
-- matches the data written. To retain simplicity, only one data cycle is
-- used (default AXI burst length encoding 0). It then initiates two write
-- data bursts, followed by two read data bursts.

library ieee;
use ieee.std_logic_1164.all;

library work;
use work.all;
use work.mgc_axi_bfm_pkg.all;

entity master_test_program is
 generic (AXI_ADDRESS_WIDTH : integer := 32;
          AXI_RDATA_WIDTH : integer := 1024;
          AXI_WDATA_WIDTH : integer := 1024;
          AXI_ID_WIDTH : integer := 4;
          index : integer range 0 to 511 :=0
         );
end master_test_program;
```

```
architecture master_test_program_a of master_test_program is
begin

  -- Master test
  process
    variable tr_id: integer;
    variable data_words         :  std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);
  begin
    wait_on(AXI_RESET_POSEDGE, index, axi_tr_if_0(index));
    wait_on(AXI_CLOCK_POSEDGE, index, axi_tr_if_0(index));

    -- 4 x Writes
    -- Write data value 1 on byte lanes 1 to address 1.
    create_write_transaction(1, tr_id, index, axi_tr_if_0(index));
    data_words(31 downto 0) := x"00000100";
    set_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    set_write_strobes(2, tr_id, index, axi_tr_if_0(index));
    report "master_test_program: Writing data (1) to address (1)";

    -- By default it will run in Blocking mode
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    -- Write data value 2 on byte lane 2 to address 2.
    create_write_transaction(2, tr_id, index, axi_tr_if_0(index));
    data_words(31 downto 0) := x"00020000";
    set_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    set_write_strobes(4, tr_id, index, axi_tr_if_0(index));
    report "master_test_program: Writing data (2) to address (2)";

    -- By default it will run in Blocking mode
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    -- Write data value 3 on byte lane 3 to address 3.
    create_write_transaction(3, tr_id, index, axi_tr_if_0(index));
    data_words(31 downto 0) := x"03000000";
    set_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    set_write_strobes(8, tr_id, index, axi_tr_if_0(index));
    report "master_test_program: Writing data (3) to address (3)";

    -- By default it will run in Blocking mode
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    -- Write data value 4 on byte lane 0 to address 4.
    create_write_transaction(4, tr_id, index, axi_tr_if_0(index));
    data_words(31 downto 0) := x"00000004";
    set_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    set_write_strobes(1, tr_id, index, axi_tr_if_0(index));
    report "master_test_program: Writing data (4) to address (4)";

    -- By default it will run in Blocking mode
    execute_transaction(tr_id, index, axi_tr_if_0(index));
```

```
    --4 x Reads
    --Read data from address 1.
    create_read_transaction(1, tr_id, index, axi_tr_if_0(index));
    set_id(1, tr_id, index, axi_tr_if_0(index));
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    get_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"00000100") then
      report "master_test_program: Read correct data (1) at address (1)";
    else
     report "master_test_program: Expected data (1) at address 1, but got
" & integer'image(to_integer(data_words(31 downto 0)));
    end if;

    --Read data from address 2.
    create_read_transaction(2, tr_id, index, axi_tr_if_0(index));
    set_id(2, tr_id, index, axi_tr_if_0(index));
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    get_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"00020000") then
      report "master_test_program: Read correct data (2) at address (2)";
    else
     report "master_test_program: Expected data (2) at address 2, but got
" & integer'image(to_integer(data_words(31 downto 0)));
    end if;

    --Read data from address 3.
    create_read_transaction(3, tr_id, index, axi_tr_if_0(index));
    set_id(3, tr_id, index, axi_tr_if_0(index));
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    get_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"03000000") then
      report "master_test_program: Read correct data (3) at address (3)";
    else
     report "master_test_program: Expected data (3) at address 3, but got
" & integer'image(to_integer(data_words(31 downto 0)));
    end if;

    --Read data from address 4.
    create_read_transaction(4, tr_id, index, axi_tr_if_0(index));
    set_id(4, tr_id, index, axi_tr_if_0(index));
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    get_data_words(data_words, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"00000004") then
      report "master_test_program: Read correct data (4) at address (4)";
    else
     report "master_test_program: Expected data (4) at address 4, but got
" & integer'image(to_integer(data_words(31 downto 0)));
    end if;
```

```
-- Write data burst length of 7 to start address 16.
create_write_transaction(16, 7, tr_id, index, axi_tr_if_0(index));
set_size(AXI_BYTES_4, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE0ACE1";
set_data_words(data_words, 0, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE2ACE3";
set_data_words(data_words, 1, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE4ACE5";
set_data_words(data_words, 2, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE6ACE7";
set_data_words(data_words, 3, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE8ACE9";
set_data_words(data_words, 4, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACEAACEB";
set_data_words(data_words, 5, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACECACED";
set_data_words(data_words, 6, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACEEACEF";
set_data_words(data_words, 7, tr_id, index, axi_tr_if_0(index));
for i in  0 to 7 loop
  set_write_strobes(15, i, tr_id, index, axi_tr_if_0(index));
end loop;
set_write_data_mode(AXI_DATA_WITH_ADDRESS, tr_id, index,
axi_tr_if_0(index));
execute_transaction(tr_id, index, axi_tr_if_0(index));

-- Write data burst length of 7 to start address 128 with
-- LSB write strobe inactive.
create_write_transaction(128, 7, tr_id, index, axi_tr_if_0(index));
set_size(AXI_BYTES_4, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE0ACE1";
set_data_words(data_words, 0, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE2ACE3";
set_data_words(data_words, 1, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE4ACE5";
set_data_words(data_words, 2, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE6ACE7";
set_data_words(data_words, 3, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE8ACE9";
set_data_words(data_words, 4, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACEAACEB";
set_data_words(data_words, 5, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACECACED";
set_data_words(data_words, 6, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACEEACEF";
set_data_words(data_words, 7, tr_id, index, axi_tr_if_0(index));
set_write_strobes(14, 0, tr_id, index, axi_tr_if_0(index));
for i in  1 to 7 loop
  set_write_strobes(15, i, tr_id, index, axi_tr_if_0(index));
end loop;
execute_transaction(tr_id, index, axi_tr_if_0(index));

-- Read data burst of length 1 from address 16.
create_read_transaction(16, 1, tr_id, index, axi_tr_if_0(index));
set_size(AXI_BYTES_4, tr_id, index, axi_tr_if_0(index));
execute_transaction(tr_id, index, axi_tr_if_0(index));
```

```
    get_data_words(data_words, 0, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"ACE0ACE1") then
      report "master_test_program: Read correct data (hACE0ACE1) at
address (16)";
    else
     report "master_test_program: Expected data (hACE0ACE1) at address
(16), but got " & integer'image(to_integer(data_words(31 downto 0)));
    end if;
    get_data_words(data_words, 1, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"ACE2ACE3") then
      report "master_test_program: Read correct data (hACE2ACE3) at
address (20)";
    else
     report "master_test_program: Expected data (hACE2ACE3) at address
(20), but got " & integer'image(to_integer(data_words(31 downto 0)));
    end if;

    -- Read data burst of length 1 from address 128.
    create_read_transaction(128, 1, tr_id, index, axi_tr_if_0(index));
    set_size(AXI_BYTES_4, tr_id, index, axi_tr_if_0(index));
    execute_transaction(tr_id, index, axi_tr_if_0(index));

    get_data_words(data_words, 0, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"ACE0AC00") then
     report "master_test_program: Read correct data (ACE0AC00) at address
(128)";
    else
     report "master_test_program: Expected data (ACE0AC00) at address
(128), but got " & integer'image(to_integer(data_words(31 downto 0)));
    end if;
    get_data_words(data_words, 1, tr_id, index, axi_tr_if_0(index));
    if(data_words(31 downto 0) = x"ACE2ACE3") then
      report "master_test_program: Read correct data (hACE2ACE3) at
address (132)";
    else
     report "master_test_program: Expected data (hACE2ACE3) at address
(132), but got " & integer'image(to_integer(data_words(31 downto 0)));
    end if;

    wait for 100 ns;
    assert false report "NONE: End of Simulation." severity failure;
  end process;
end master_test_program_a;
```

VHDL Slave BFM Code Example

```
*******************************************************************************
--
-- Copyright 2007-2012 Mentor Graphics Corporation
-- All Rights Reserved.
--
-- THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
-- WHICH IS THE PROPERTY OF
-- MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT
-- TO LICENSE TERMS.
--
*******************************************************************************
--
-- This simplfied example of an AXI Slave demonstrates using the
-- mgc_axi_slave BFM. This fairly generic slave handles almost all write
-- and read transaction scenarios from the master. It handles both, write
-- data with address, as well as data after the address.
--
-- This slave code is divided in two parts, one which the user might need
-- to edit to change the slave mode(for example, for Transaction/burst or
-- Phase level) and memory handling.
-- Other code, which is labeled, "user need not edit", can be edited if
-- user needs to meet required phase valid/ready delays.
--
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.all;
use work.mgc_axi_bfm_pkg.all;

entity slave_test_program is
   generic (AXI_ADDRESS_WIDTH : integer := 32;
            AXI_RDATA_WIDTH : integer := 1024;
            AXI_WDATA_WIDTH : integer := 1024;
            AXI_ID_WIDTH : integer := 4;
            index : integer range 0 to 511 := 0
            );
 end slave_test_program;

architecture slave_test_program_a of slave_test_program is
  type axi_slave_mode_e is (AXI_TRANSACTION_SLAVE, AXI_PHASE_SLAVE);
  type memory_t is array (0 to 2**16-1) of std_logic_vector(7 downto 0);
  shared variable mem : memory_t;
```

```
  signal slave_mode : axi_slave_mode_e := AXI_TRANSACTION_SLAVE;
  procedure do_byte_read(addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0); data : out std_logic_vector(7 downto 0));
  procedure do_byte_write(addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);  data : in std_logic_vector(7 downto 0));
  procedure set_write_address_ready_delay(id : integer; signal tr_if :
inout axi_vhd_if_struct_t);
  procedure set_write_address_ready_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t);
  procedure set_read_address_ready_delay(id : integer; signal tr_if :
inout axi_vhd_if_struct_t);
  procedure set_read_address_ready_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t);
  procedure set_write_data_ready_delay(id : integer; signal tr_if : inout
axi_vhd_if_struct_t);
  procedure set_write_data_ready_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t);
  procedure set_wr_resp_valid_delay(id : integer; signal tr_if : inout
axi_vhd_if_struct_t);
 procedure set_wr_resp_valid_delay(id : integer; path_id : in axi_path_t;
signal tr_if : inout axi_vhd_if_struct_t);
  procedure set_read_data_valid_delay(id : integer; signal tr_if : inout
axi_vhd_if_struct_t);
  procedure set_read_data_valid_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t);

  -- Procedure: do_byte_read
  -- This procedure specifies the read data byte wil be read from memory at
  -- a particular input address.
  procedure do_byte_read(addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0); data: out std_logic_vector(7 downto 0)) is
  begin
    data := mem(to_integer(addr));
  end do_byte_read;

  -- Procedure: do_byte_write
  -- This procedure writes a data byte to memory at a particular input
  -- address.
  procedure do_byte_write(addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);  data : in std_logic_vector(7 downto 0)) is
  begin
    mem(to_integer(addr)) := data;
  end do_byte_write;

  -- Procedure: set_write_address_ready_delay
  -- This procedure sets and extends the write address phase ready delay
  -- phase.
  procedure set_write_address_ready_delay(id : integer; signal tr_if :
inout axi_vhd_if_struct_t) is
  begin
    set_address_ready_delay(2, id, index, tr_if);
  end set_write_address_ready_delay;
  procedure set_write_address_ready_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t) is
  begin
    set_address_ready_delay(2, id, index, path_id, tr_if);
  end set_write_address_ready_delay;
```

```
  -- Procedure: set_read_address_ready_delay
  -- This procedure sets and extends the read address phase ready delay.
  procedure set_read_address_ready_delay(id : integer; signal tr_if :
inout axi_vhd_if_struct_t) is
  begin
    set_address_ready_delay(4, id, index, tr_if);
  end set_read_address_ready_delay;
  procedure set_read_address_ready_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t) is
  begin
    set_address_ready_delay(4, id, index, path_id, tr_if);
  end set_read_address_ready_delay;

  -- Procedure: set_write_data_ready_delay
  -- This procedure sets the ready delays for each write data phase in a
  --write data burst.
  procedure set_write_data_ready_delay(id : integer; signal tr_if : inout
axi_vhd_if_struct_t) is
    variable burst_length : integer;
  begin
    get_burst_length(burst_length, id, index, tr_if);
    for i in  0 to burst_length loop
      set_data_ready_delay(i, i, id, index, tr_if);
    end loop;
  end set_write_data_ready_delay;
  procedure set_write_data_ready_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t) is
    variable burst_length : integer;
  begin
    get_burst_length(burst_length, id, index, path_id, tr_if);
    for i in  0 to burst_length loop
      set_data_ready_delay(i, i, id, index, path_id, tr_if);
    end loop;
  end set_write_data_ready_delay;

  -- Procedure: set_wr_resp_valid_delay
  -- This procedure sets the write response phase valid delay to start
  -- driving write response phase after the specified delay.
  procedure set_wr_resp_valid_delay(id : integer; signal tr_if : inout
axi_vhd_if_struct_t) is
  begin
    set_write_response_valid_delay(0, id, index, tr_if);
  end set_wr_resp_valid_delay;
 procedure set_wr_resp_valid_delay(id : integer; path_id : in axi_path_t;
signal tr_if : inout axi_vhd_if_struct_t) is
  begin
    set_write_response_valid_delay(0, id, index, path_id, tr_if);
  end set_wr_resp_valid_delay;
```

```
  -- Procedure: set_read_data_valid_delay
  -- This procedure sets the ready delays for each write data phase in a
  --write data burst.
  procedure set_read_data_valid_delay(id : integer; signal tr_if : inout
axi_vhd_if_struct_t) is
    variable burst_length : integer;
  begin
    get_burst_length(burst_length, id, index, tr_if);
    for i in  0 to burst_length loop
      set_data_valid_delay(i, i, id, index, tr_if);
    end loop;
  end set_read_data_valid_delay;
  procedure set_read_data_valid_delay(id : integer; path_id : in
axi_path_t; signal tr_if : inout axi_vhd_if_struct_t) is
    variable burst_length : integer;
  begin
    get_burst_length(burst_length, id, index, path_id, tr_if);
    for i in  0 to burst_length loop
      set_data_valid_delay(i, i, id, index, path_id, tr_if);
    end loop;
  end set_read_data_valid_delay;

begin

  -- To create pipelining in VHDL there are multiple channel paths in each
  -- API. Each process chooses a separate path to interact with the BFM.
  -- process_write: write address phase through path 0
  -- This process continuously receives write address phase and pushes
  -- the transaction into the queue through the push_transaction_id API.
  process
    variable write_trans : integer;
  begin
    wait for 1ns;
    wait_on(AXI_RESET_POSEDGE, index, axi_tr_if_0(index));
    loop
      create_slave_transaction(write_trans, index, axi_tr_if_0(index));
      set_write_address_ready_delay(write_trans, axi_tr_if_0(index));
      get_write_addr_phase(write_trans, index, axi_tr_if_0(index));
      push_transaction_id(write_trans, AXI_QUEUE_ID_0, index,
axi_tr_if_0(index));
    end loop;
    wait;
  end process;
```

```
  -- handle_write: writes data phase through path 1
  -- This method receives the write data burst or phases for write
  -- transactions based on the slave working mode and the write data to
  -- memory.
  process
    variable write_trans: integer;
    variable byte_length : integer;
    variable burst_length : integer;
    variable addr : std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
    variable data : std_logic_vector(7 downto 0);
    variable last : integer := 0;
    variable loop_i : integer := 0;
  begin
    loop
      pop_transaction_id(write_trans, AXI_QUEUE_ID_0, index, AXI_PATH_1,
axi_tr_if_1(index));
      set_write_data_ready_delay(write_trans, AXI_PATH_1,
axi_tr_if_1(index));

      if (slave_mode = AXI_TRANSACTION_SLAVE) then
        get_write_data_burst(write_trans, index, AXI_PATH_1,
axi_tr_if_1(index));
        get_burst_length(burst_length, write_trans, index, AXI_PATH_1,
axi_tr_if_1(index));
        for i in  0 to burst_length loop
         get_write_addr_data(write_trans,  i, 0, byte_length, addr, data,
index, AXI_PATH_1, axi_tr_if_1(index));
          do_byte_write(addr, data);
          if byte_length > 1 then
            for j in 1 to byte_length-1 loop
               get_write_addr_data(write_trans,  i, j, byte_length, addr,
data, index, AXI_PATH_1, axi_tr_if_1(index));
               do_byte_write(addr, data);
            end loop;
          end if;
        end loop;
      else
        last := 0;
        loop_i := 0;
        while(last = 0) loop
          get_write_data_phase(write_trans, loop_i, last, index,
AXI_PATH_1, axi_tr_if_1(index));
          loop_i := loop_i + 1;
         get_write_addr_data(write_trans,  loop_i, 0, byte_length, addr,
data, index, AXI_PATH_1, axi_tr_if_1(index));
          do_byte_write(addr, data);
          if byte_length > 1 then
            for j in 1 to byte_length-1 loop
               get_write_addr_data(write_trans,  loop_i, j, byte_length,
addr, data, index, AXI_PATH_1, axi_tr_if_1(index));
               do_byte_write(addr, data);
            end loop;
          end if;
        end loop;
      end if;
      push_transaction_id(write_trans, AXI_QUEUE_ID_2, index, AXI_PATH_1,
axi_tr_if_1(index));
    end loop;
```

```
    wait;
  end process;

  -- handle_response: writes response phase through path 2
  -- This method sends the write response phase
  process
    variable write_trans: integer;
  begin
    loop
      pop_transaction_id(write_trans, AXI_QUEUE_ID_2, index, AXI_PATH_2,
axi_tr_if_2(index));
      set_wr_resp_valid_delay(write_trans, AXI_PATH_2,
axi_tr_if_2(index));
      execute_write_response_phase(write_trans, index, AXI_PATH_2,
axi_tr_if_2(index));
    end loop;
    wait;
  end process;

  -- process_read: reads the address phase through path 3
  -- This process contiuooously receives the read address phase and pushes
  -- the transaction into queue through the push_transaction_id API.
  process
    variable read_trans: integer;
  begin
    wait for 1ns;
    wait_on(AXI_RESET_POSEDGE, index, AXI_PATH_3, axi_tr_if_3(index));
    loop
      create_slave_transaction(read_trans, index, AXI_PATH_3,
axi_tr_if_3(index));
      set_read_address_ready_delay(read_trans, AXI_PATH_3,
axi_tr_if_3(index));
      get_read_addr_phase(read_trans, index, AXI_PATH_3,
axi_tr_if_3(index));
      push_transaction_id(read_trans, AXI_QUEUE_ID_1, index, AXI_PATH_3,
axi_tr_if_3(index));
    end loop;
    wait;
  end process;

  -- handle_read: reads the data and responds through path 4
  -- This process reads data from memory and sends the read data/response
  -- either at the burst or phase level, based on the slave mode.
  process
    variable read_trans: integer;
    variable burst_length : integer;
    variable byte_length : integer;
    variable addr : std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
    variable data : std_logic_vector(7 downto 0);
  begin
    loop
      pop_transaction_id(read_trans, AXI_QUEUE_ID_1, index, AXI_PATH_4,
axi_tr_if_4(index));
      set_read_data_valid_delay(read_trans, AXI_PATH_4,
axi_tr_if_4(index));
```

```
      get_burst_length(burst_length, read_trans, index, AXI_PATH_4,
axi_tr_if_4(index));
      for i in  0 to burst_length loop
        get_read_addr(read_trans,  i, 0, byte_length, addr, index,
AXI_PATH_4, axi_tr_if_4(index));
        do_byte_read(addr, data);
        set_read_data(read_trans, i, 0, byte_length, addr, data, index,
AXI_PATH_4, axi_tr_if_4(index));
        if slave_mode = AXI_PHASE_SLAVE then
          execute_read_data_phase(read_trans, i, index, AXI_PATH_4,
axi_tr_if_4(index));
        end if;
        if byte_length > 1 then
          for j in 1 to byte_length-1 loop
             get_read_addr(read_trans,  i, j, byte_length, addr, index,
AXI_PATH_4, axi_tr_if_4(index));
             do_byte_read(addr, data);
             set_read_data(read_trans, i, j, byte_length, addr, data,
index, AXI_PATH_4, axi_tr_if_4(index));
             if slave_mode = AXI_PHASE_SLAVE then
               execute_read_data_phase(read_trans, i, index, AXI_PATH_4,
axi_tr_if_4(index));
             end if;
          end loop;
        end if;
      end loop;
      if slave_mode = AXI_TRANSACTION_SLAVE then
        execute_read_data_burst(read_trans, index, AXI_PATH_4,
axi_tr_if_4(index));
      end if;
    end loop;
    wait;
  end process;
end slave_test_program_a;
```

Third-party Software for Mentor Verification IP Altera Edition

This section provides information on open source and third-party software that may be included in the Mentor Verification IP Altera Edition software product.

This software application may include GNU GCC 4.3.3 third-party software. GNU GCC v4.3.3 is distributed under the terms of the GNU General Public License version 3.0 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <install_directory>/docs/legal/gnu_gpl_3.0.pdf. Portions of this software may be subject to the GNU Free Documentation License v1.2. You can view a copy of the GNU Free Documentation License v1.2 at: <install_directory>/docs/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the GNU Lesser General Public License v2.1. You can view a copy of the GNU Lesser General Public License v2.1 at: <install_directory>/docs/legal/ gnu_lgpl_2.1.pdf. Portions of this software may be subject to the GNU Library General Public License v2. You can view a copy of the GNU Library General Public License v2 at: <install_directory>/docs/legal/ gnu_library_gpl_2.0.pdf. Portions of this software may be subject to the W3C License. You can view a copy of the W3C License at: <install_directory>/docs/legal/w3c_2002.pdf. Portions of this software may be subject to the Boost License version 1.0. You can view a copy of the Boost License v1.0 at: <install_directory>/docs/legal/ boost_1.0.pdf. To obtain a copy of the GNU GCC v4.3.3 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed GNU GCC v4.3.3 and valid for as long as Mentor Graphics offers customer support for this Mentor Graphics product. GNU GCC v4.3.3 may be subject to the following copyrights:

© 1987 Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

© 1983, 1990, 1991 Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. [rescinded 22 July 1999]

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT

SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1991, 2000, 2006 AT&T
The author of this software is David M. Gay.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

© 2004 World Wide Web Consortium

(Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. This work is distributed under the W3C(r) Software License [1] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```
[1] http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231
```

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications:

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.

2. Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C Software Short Notice should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code.

3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

© 1996-1999 Silicon Graphics Computer Systems, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1994, 2002 Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1997 Christian Michelsen Research AS Advanced Computing
Fantoftvegen 38, 5036 BERGEN, Norway
http://www.cmr.no

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Christian Michelsen Research AS makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 2000-2005 INRIA, France Telecom
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 2001 Thai Open Source Software Center Ltd.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the Thai Open Source Software Center Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 2004, Eugene Kuleshov
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1991-2005 Unicode, Inc.
All rights reserved.

Distributed under the Terms of Use in http://www.unicode.org/copyright.html

Permission is hereby granted, free of charge, to any person obtaining a copy of the Unicode data files and any associated documentation (the "Data Files") or Unicode software and any associated documentation (the "Software") to deal in the Data Files or Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Data Files or Software, and to permit persons to whom the Data Files or Software are furnished to do so, provided that (a) the above copyright notice(s) and this permission notice appear with all copies of the Data Files or Software, (b) both the above copyright notice(s) and this permission notice appear in associated documentation, and (c) there is clear notice in each modified Data File or in the Software as well as in the documentation associated with the Data File(s) or Software that the data or software has been modified.

THE DATA FILES AND SOFTWARE ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE DATA FILES OR SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in these Data Files or Software without prior written authorization of the copyright holder.

© 2004 Ami Tavory and Vladimir Dreizin, IBM-HRL

Permission to use, copy, modify, sell, and distribute this software is hereby granted without fee, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation. None of the above authors, nor IBM Haifa Research Laboratories, make any representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1997 - 2002, Makoto Matsumoto and Takuji Nishimura
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1999 Citrus Project
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software application may include Binutils version 2.19.1 third-party software. Binutils v2.19.1 is distributed under the terms of the GNU General Public License version 3.0 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_gpl_3.0.pdf. Portions of this software may be subject to the GNU Free Documentation License v1.2. You can view a copy of the GNU Free Documentation License v1.2 at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the GNU Library General Public License v2. You can view a copy of the GNU Library General Public License v2 at: <your_Mentor_Graphics_documentation_directory>/legal/ gnu_library_gpl_2.0.pdf. Portions of this software may be subject to the GNU Lesser General Public License v3.0. You can view a copy of the GNU Lesser General Public License v3.0 at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_lgpl_3.0.pdf. To obtain a copy of the Binutils v2.19.1 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed Binutils v2.19.1 and valid for as long as Mentor Graphics offers customer support for this Mentor Graphics product. Binutils v2.19.1 may be subject to the following copyrights:

© 1983, 1990, 1991, 1993, 1998, 2001, 2002 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1987 Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California,

Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

© 1997 John D. Polstra All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1998 Cygnus Solutions.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies. Permission is granted to process this file through TeX and print the results, provided the printed document carries copying permission notice identical to this one except for the removal of this paragraph Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one. Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

This software application may include MinGW gcc version 4.2.1 third-party software. MinGW gcc version 4.2.1 is distributed under the terms of the GNU General Public License version 2.0 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_gpl_2.0.pdf. Portions of this software may be subject to the GNU Free Documentation License v1.2. You can view a copy of the GNU Free Documentation License v1.2 at: <your_Mentor_Graphics_documentation_directory>/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the GNU Lesser General Public License v2.1. You can view a copy of the GNU Lesser General Public License v2.1 at: <your_Mentor_Graphics_documentation_directory>/legal/ gnu_lgpl_2.1.pdf . To obtain a copy of the MinGW gcc version 4.2.1 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed MinGW gcc version 4.2.1. MinGW gcc version 4.2.1 may be subject to the following copyrights:

© 1994

Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1996,1997, 1998

Silicon Graphics Computer Systems, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 2004 Ami Tavory and Vladimir Dreizin, IBM-HRL.

Permission to use, copy, modify, sell, and distribute this software is hereby granted without fee, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation. None of the above authors, nor IBM Haifa Research Laboratories, make any representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1982, 1986, 1992, 1993

The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

   This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1992, 1993, 1994 Henry Spencer. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.

2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.

3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.

4. This notice may not be removed or altered.

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **ORDERS, FEES AND PAYMENT.**

   1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement ("Order(s)"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.

   1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

   1.3. All Products are delivered FCA factory (Incoterms 2000), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer requests any change or enhancement to Software, whether in the course of receiving support or consulting services, evaluating Software, performing beta testing or otherwise, any inventions, product

improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

   4.1. Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

   4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

   4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

   5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use it except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Software or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Software, or disclose to any third party the results of, or information pertaining to, any benchmark.

   5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

   5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at http://supportnet.mentor.com/about/legal/.

7. **AUTOMATIC CHECK FOR UPDATES; PRIVACY.** Technological measures in Software may communicate with servers of Mentor Graphics or its contractors for the purpose of checking for and notifying the user of updates and to ensure that the Software in use is licensed in compliance with this Agreement. Mentor Graphics will not collect any personally identifiable data in this process and will not disclose any data collected to any third party without the prior written consent of Customer, except to Mentor Graphics' outside attorneys or as may be required by a court of competent jurisdiction.

8. **LIMITED WARRANTY.**

8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **INDEMNIFICATION.** CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10. THE PROVISIONS OF THIS SECTION 11 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

12. **INFRINGEMENT.**

12.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

12.2. If a claim is made under Subsection 12.1 Mentor Graphics may, at its option and expense, (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

12.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

12.4. THIS SECTION 12 IS SUBJECT TO SECTION 9 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS FOR DEFENSE, SETTLEMENT AND DAMAGES, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

13. **TERMINATION AND EFFECT OF TERMINATION.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term.

13.1. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

13.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

14. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products and information about the products to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

15. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

16. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

17. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 17 shall survive the termination of this Agreement.

18. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the United States. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, USA, if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. This section shall not

restrict Mentor Graphics' right to bring an action against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

19. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

20. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 100615, Part No. 246066